Adaptive Automated Machine Learning

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften
(Dr.-Ing.)

von der KIT-Fakultät für
Wirtschaftswissenschaften
des Karlsruher Instituts für Technologie (KIT)

genehmigte
DISSERTATION

von

M.Sc. Cedric Peter Charles Kulbach

Tag der mündlichen Prüfung:	08.09.2022
Referent:	Prof. Dr. York Sure-Vetter
Korreferent:	Prof. Dr. Albert Bifet

Karlsruhe 2022

Abstract

The ever-growing demand for machine learning has led to the development of automated machine learning (AutoML) systems that can be used off the shelf by non-experts. Further, the demand for ML applications with high predictive performance exceeds the number of machine learning experts and makes the development of AutoML systems necessary. Automated Machine Learning tackles the problem of finding machine learning models with high predictive performance. Existing approaches incorporating deep learning techniques assume that all data is available at the beginning of the training process (offline learning). They configure and optimise a pipeline of preprocessing, feature engineering, and model selection by choosing suitable hyperparameters in each model pipeline step. Furthermore, they assume that the user is fully aware of the choice and, thus, the consequences of the underlying metric (such as precision, recall, or F1-measure). By variation of this metric, the search for suitable configurations and thus the adaptation of algorithms can be tailored to the user's needs. With the creation of a vast amount of data from all kinds of sources every day, our capability to process and understand these data sets in a single batch is no longer viable. By training machine learning models incrementally (i.ex. online learning), the flood of data can be processed sequentially within data streams. However, if one assumes an online learning scenario, where an AutoML instance executes on evolving data streams, the question of the best model and its configuration remains open.

In this work, we address the adaptation of AutoML in an offline learning scenario toward a certain utility an end-user might pursue as well as the adaptation of AutoML towards evolving data streams in an online learning scenario with three main contributions:

1. We propose a System that allows the adaptation of AutoML and the search for neural architectures towards a particular utility an end-user might pursue.

2. We introduce an online deep learning framework that fosters the research of deep learning models under the online learning assumption and enables the automated search for neural architectures.

3. We introduce an online AutoML framework that allows the incremental adaptation of ML models.

We evaluate the contributions individually, in accordance with predefined requirements and to state-of-the-art evaluation setups. The outcomes lead us to conclude that (i) AutoML, as well as systems for neural architecture search, can be steered towards individual utilities by learning a designated ranking model from pairwise preferences and using the latter as the target function for the offline learning scenario; (ii) architectual small neural networks are in general suitable assuming an online learning scenario; (iii) the configuration of machine learning pipelines can be automatically be adapted to ever-evolving data streams and lead to better performances.

Contents

Part I

Overview

1 Introduction

Given the topic *"Adaptive Automated Machine Learning"*, we motivate this thesis by starting with the need for automation of commonly consciously or unconsciously practised Data Mining processes. We introduce the problem of adaptivity in *AutoML* by first motivating the setting in Section 1.1 and presenting the challenges in Section 1.2. We then introduce our hypotheses and research questions in Section 1.3. In Section 1.4, we frame the scope and the contributions of this work by pointing out which aspects we focus on and which we neglect. In the last section of this introduction, Section 1.5, we give an overview of all parts and chapters of this thesis.

1.1 Motivation

Already in 1989 Frawley et al. [86] claimed that the amount of information in the world doubles every 20 months. Information that is stored in millions of databases that describe potentially valuable patterns in hidden ways. To cope with this amount of data, massive research has been employed towards a standardised process for mining these patterns. The term most commonly employed for this purpose is *KDD*, which was coined in 1989 by Piatetsky-Shapiro. As the name states, the *KDD* process implements the process of discovering valuable knowledge (patterns) from data [79] and thus assists humans in extracting useful information in an environment of exploding volumes of data. Based on the *KDD* process, extensions such as the *CRISP-DM* [50] further standardised and extended the process of *DM*. Besides the introduction of *KDD* and its implementation *CRISP-DM*, Piatetsky-Shapiro [190] further claimed the success of *ML* in this area to be capable of dealing with the amounts of data:

> "The next area that is going to explode is the use of machine learning tools as a component of large scale data analysis."
>
> –Piatetsky-Shapiro, 1989

It is superfluous to say that nowadays, we face a flood of data from all kinds of applications, devices, and different formats containing valuable information. And even further, we can foresee that *IoT* and *IIoT* applications even raise the scale of data to an unprecedented level [23]. To be capable of dealing with this amount of data, the development of a vast number of *ML* tools and applications took place, and its success in a broad range of applications has led to an ever-growing demand for *ML* systems. As a result, similar to the development towards a standardised *DM* and *KDD* processes, research in *ML* led to a common iteration of steps to successfully employ *ML* models within a *ML* pipeline. The increasing computing power at our disposal led, further, to the application of more and more complex models such as *NN* in the field of *DL*. In addition to standardising *ML* pipelines, *DL* has shown its strong ability to extract new previously unknown patterns from data. However, the common development of *ML* applications that successfully

decode knowledge from data follows in accordance with the *KDD* process the exact development steps. Still, it requires experts such as data scientists that employ and configure *DM* pipelines that include *ML* steps.

To meet the demand for *ML* systems, *AutoML* and *NAS* frameworks aim at automatically propose suited *ML* pipelines or *DL* models that are automaticly orchestrated and configured by following a predefined target. *AutoML*, however, is the challenge of finding *ML* pipelines with high predictive performance without the need for specialised data scientists and thus theoretically incorporates the search for suitable *DL* models, which is denoted to as *NAS*. Existing approaches optimise a pipeline of preprocessing, feature engineering, model selection and hyperparameter optimisation and thus cover a few steps of classical *DM* processes. However, to be effective in practice, such systems choose based on a predefined target good algorithm and feature preprocessing steps for a given data set and set their respective hyperparameters. For this purpose, they have shown impressive results in learning patterns and thus performing classification or regression tasks accordingly to *KDD* given a predefined objective and when assuming that all data is available at once. Due to their different characteristics and tasks they aim to solve, *AutoML* and *NAS* are usually considered separately.

A particular drawback of today's *AutoML* frameworks is their lack of adaptability:
They assume an awareness of the underlying metric that follows a target. This metric is defined beforehand, does not essentially adapt to the user's utility and thus may not direct *AutoML* and *NAS* systems towards a desired solution. Further, they are based on the data at hand at the beginning of the process and thus assume that data patterns do not change over time. *AutoML* has shown impressive performance on offline learning tasks in which all data is available at once. However, many real-world environments generate data continuously and indefinitely in the form of never-ending data streams [23, 92]. Considering i.ex. an *IoT* or *IIoT* environment, data is often produced as a data stream, i.ex; sensors produce over time floods of data, where the environment and thus the patterns may change. Consequently, the system is challenged to adapt.

This work investigates the extension of *AutoML* including *NAS* techniques towards their adaptivity. Whereby the term adaptivity is two-folded, as, on the one hand, *AutoML* and *NAS* techniques are adapted towards the utility a user might pursue and, on the other hand, towards their ability to adapt to changes within the data patterns that often occur in data streams.

1.2 Challenges

This section describes the challenges and problems we investigate in this work towards Adaptive Automated Machine Learning, including *Neural Architecture Search*. We refer to Chapter 2 for an in-depth introduction to *AutoML* and *NAS* starting with the *KDD* process. In Figure 1.1, we locate the utility adaptation of *AutoML* and the adaptation to data streams on the *CRISP-DM* process. As illustrated in Figure 1.1, *AutoML* and thus

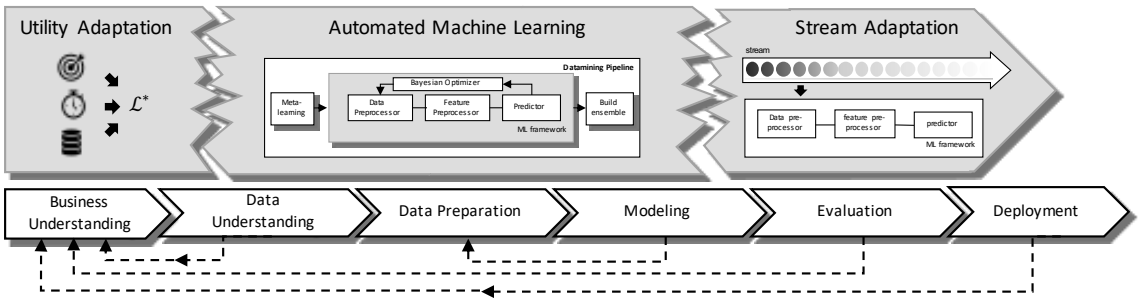


Figure 1.1: Illustration of connecting principles towards a utility and pattern based adaptation of *AutoML* systems in the context of the *CRISP-DM* [50] process.

also *NAS* frameworks only cover few steps of the depicted *CRISP-DM* process exemplary illustrated for a *DM* pipeline. Notably, they adapt without an understanding of the underlying business and consideration of a continuous evaluation protocol. While *AutoML* tries to automate a *ML* pipeline by employing preprocessing, feature engineering, model selection and hyperparameter optimisation, classical *DM* approaches such as the *CRISP-DM* process go beyond these pipeline building steps. Referring to the beginning of the *CRISP-DM* process in Figure 1.1, *AutoML* does not cover the business understanding step since *AutoML* frameworks require the awareness of an underlying metric that the *AutoML* system should pursue to extract patterns and thus to perform a task such as classification or regression. Further, the search for suitable neural architectures poses new problems despite the similarity to *AutoML*. In comparison, *NAS* systems are more complex due to an immense and theoretically infinite ample configuration space and further to their computation complexity during training. While *AutoML* systems take depending on the data and the available computing power several hours [177] to find suitable *ML* pipelines, *NAS* can take several weeks [198] to train and find suitable neural architectures. In Section 1.2.1 we investigate on the problem statement towards an utility-based adaptation of *AutoML* and *NAS* systems. By referring to the last steps of the *CRISP-DM* process, the steps that consider the evaluation and deployment, commonly applied *AutoML* systems optimise the underlying *ML* pipeline based on a given data set and are thus evaluated on data that was available at a given point. Already in this setting, it emerges that the validity of the evaluation is limited to the collected data set. However, considering that underlying patterns and data distributions change over time and data is continuously generated not only in *IIoT* or *IoT* environments, it raises the problem of adapting the *AutoML* or *NAS* system to new data that may contain changing patterns. Thus the challenge within the adaptation of *AutoML* or *NAS* systems is to find suitable *ML* pipelines or neural architectures that are not limited to be ideal for a given data set at a certain point; they need to adapt constantly to continuously available data in the form of evolving data streams. In Section 1.2.2, we investigate on the problem towards a stream based adaptation of *AutoML* and *NAS* systems.

1.2.1 Utility Based Adaptation

Current approaches for *AutoML* and *NAS* aim at efficiently maximising individual or sets of objectives. For this purpose, they need a preset metric (e.g. accuracy) to optimise the underlying *ML* pipeline or neural architecture. However, end-users objectives are diverse, reaching beyond the accuracy of a classification task. The end-user is, thus, assumed to be fully aware of his or her preference and knows the available metrics that are suitable to represent this preference. To find suitable solutions, current research considers only a few metric functions (e.g. precision, recall) optimised within a Pareto-frontier, which remains unscalable with a more significant number of objectives. Further, they do not explore sets of metrics that inherently follow the same goal (e.g. optimising predictive performance with precision, recall, accuracy and F1-measure).

The utility-based adaptation of *AutoML* and *NAS* aims to close this gap by learning the user's preference and steering the optimisation process of the *AutoML* or *NAS* system into the direction an end-user or domain expert might pursue. As illustrated in Figure 1.1, a utility-based metric $\mathcal{L}^*$ could incorporate metrics based on the predictive performance but also time and memory relevant metrics. This research field, however, is closely related to the research fields *HGML* and *metric learning*. While the goal of *HGML* is to create ". . . systems that allow a domain expert, without a machine learning expert, to use relevant domain knowledge to inform the automated search for high quality, impactful and interpretable model, including necessary data preparation steps necessary for analysis" [95], the goal of *metric learning* is to learn a data-dependent metric that measures the performance of a distance-related (*ML*) model [150] (see Section 3.1). Derived from both research areas [95, 150], we can set the following main requirements for a utility-based *AutoML* and *NAS* framework and the underlying utility metric.

R II–1 An end-user may give certain variables and parameters of the underlying model more priority.

R II–2 The end-users utility should reflect within the metric.

R II–3 A utility-based metric should influence the optimisation of an underlying model in the direction of the utility

R II–4 The utility-metric should follow a symmetry.

R II–5 The metric should be non-negative.

The first requirement describes that an end-user should be able to state their preference towards their utility. This requirement is manifold in that its fulfilment is dependent on the set of parameters; from the method, certain variables and parameters are preferred, and from the interface, the user interacts with to state his or her preference. As a set of parameters, one could consider a broad range of already established metrics, such as accuracy, latency, precision, recall, number of parameters of the underlying model etc., that emerged to be valuable over time. As a method to state the user's preference, we base our evaluation on an approach that learns the underlying utility by pairwise comparisons, whereby we exclude the user interface from the evaluation as it entails too numerous dependencies that influence the evaluation. The third requirement (*R II–2*) concerns the (*metric learning*) method that should reflect the underlying utility and further it should steer the optimisation process (*R II–3*) toward this utility. Further, the utility should follow a symmetry (*R II–4*) that provides the same metric score when the function's input is switched, and the utility metric should be non-negative (*R II–5*). The utility-based adaptation of *AutoML* and *NAS* thus pose the problem of how to teach the model a utility that goes beyond predictive performance.

1.2.2 Stream Based Adaptation

Considering offline learning tasks in which the entire data set is available at once, *AutoML* and *NAS* systems have shown impressive results by configuring pipelines of algorithms or the architectures of a *NN*s. However, many real-world environments generate data continuously and infinitely in the form of never-ending data streams [23, 92]. Unlike the offline setting, the unbounded nature of these data raises some practical and technical requirements that need to be addressed, where streaming algorithms follow the online learning requirements defined by Bifet et al. [23]:

R I–1 Process an instance at a time and inspect it (at most) once.

R I–2 Use a limited amount of time to process each instance.

R I–3 Use a limited amount of memory.

R I–4 Be ready to give an answer (e.g. prediction) at any time.

R I–5 Adapt to temporal changes.

The first requirement refers to *incremental learning*, where the data stream is processed iteratively. The requirement towards a time and memory limitation of the underlying algorithm, as well as the availability to give an answer at any time, results from the nature of data streams. This is especially the case in *IoT* and *IIoT* environments, where data is produced commonly in high volume and high frequency and thus requires efficient processing to guarantee a certain throughput and thus to be ready to predict at any time (*R I–4*). The last requirement defined by Bifet et al. refers to the adaptability to temporal changes of the underlying model. However, when retraining *AutoML* or other offline learning algorithms where all data is at hand at once, a significant part of these requirements is infringed. Referring to Figure 1.1, the problem towards the adaption to data streams and thus to changing patterns is located at the end of the *CRISP-DM* process,

where a further question arises about the evaluation of *ML* algorithms in general and for the scope of this work about the evaluation of *AutoML* and *NAS* systems in dynamic environments. The evaluation step, as illustrated in Figure 1.1, cannot be seen separately from the *CRISP-DM* process, since the question about the evaluation influences the answer to the question for the best *ML* pipeline in *AutoML* and for the best neural architecture in *NAS*. The question about the evaluation, thus, incorporates the model's configuration and hence influences the optimisation process of the entire *AutoML* and *NAS* system. The challenge within the adaptation of *AutoML* or *NAS* to data streams is thus to find suitable *ML* pipelines or neural architectures within a dynamic data stream environment.

1.3 Hypotheses & Research Questions

Our research aims to provide novel methodologies that steer data-centric *AutoML* and *NAS* frameworks towards adaptivity. We approach this adaptivity from two sides. On the one hand, from the adaptation to a particular utility, an end-user might pursue and on the other hand, from a dynamic environment, where data becomes continuously available in the form of data streams. Targeting the adaptation towards a utility-based *AutoML* and *NAS* framework we assume that (i) the target function $\mathcal{L}$ can be modified into the direction an end-user might pursue and (ii) that the variation of the target function has an influence on the output of *AutoML* and *NAS* frameworks.

RQ I. How can an *AutoML* system be adapted to a utility an end-user might pursue?

RQ I.1. How can a new target function $\mathcal{L}^*$ be learned?

RQ I.2. How can we optimise an *AutoML* system towards the learned utility?

The first research question RQ I, thus asks the adaptivity of *AutoML* systems to a certain utility. We split RQ I into the research questions RQ I.1 and RQ I.2, where the first research question is about the variation of a target function and how to express the utility an end-user could pursue within a target function $\mathcal{L}$. RQ I.2 asks for the integration of the adapted target function $\mathcal{L}^*$ into an *AutoML* system and its influence on the performance towards the utility.

RQ II. How can a *NAS* system be adapted to the user's utility?

RQ II.1. How can a new target function $\mathcal{L}^*$ beyond predictive performance measurements be learned?

RQ II.2. How does a learned target function influence *NAS* towards the pursued utility?

Despite the fact, that *NAS* can be seen as specialisation of *AutoML*, *NAS* poses new problems in terms of configuration and computational complexity. Thus, in this thesis, *NAS* is seen separately to *AutoML*. Following RQ I we ask in RQ II for the adaptivity of *NAS* systems. The greater complexity of *NAS* systems requires even more attention to metrics that go beyond predictive performance, and thus the question arises how new target functions $\mathcal{L}^*$ that go beyond the predictive performance can be learned (RQ II.1). Further,

the larger (possibly infinite) configuration space raises the question RQ II.2 towards the impact of a learned target $\mathcal{L}^*$. As we assume that the target function $\mathcal{L}$ can be modified into the direction of the utility and the learned metric influences the *AutoML* and *NAS* framework into the direction of the utility, we derive based on the presented research question Hypothesis I.

Hypothesis I (Utility Adaptation)
Existing approaches for AutoML *and* NAS *aim efficiently maximising individual or sets of objectives $\mathcal{L}$. By variation of the target function $\mathcal{L}$, the output of* AutoML *systems can be adapted and tailored to the needs of the user and thus to a utility.*

Targeting the adaptation of *AutoML* and *NAS* to data streams and thus changing patterns we assume that (i) the configuration of *AutoML* and *NAS* systems is possible accordingly to the requirements defined by Bifet et al. [23] in an incremental manner and that the incremental adaptation (ii) enables better performances in form of the adaptation to potentially infinite data streams and changing patterns of *AutoML* and *NAS* systems. Based on the problem description presented in Section 1.2.2, we derive the following research questions.

RQ III. How can *HPO* techniques be applied in an online learning environment to further enable online *AutoML* and *NAS*?

RQ III.1. Are neural networks suitable for online learning?

RQ III.2. How can the hyperparameters of *ML* pipelines (*AutoML*) and *NN*s (*NAS*) incrementally be adapted to data streams?

RQ III.3. Does an incremental adaptation of hyperparameters in *AutoML* or *NAS* systems enable better performances on data streams?

The main research question (RQ III) asks for the adaptation of *AutoML* systems to data streams and thus for an *HPO* technique capable of handling data streams. However, to apply *NAS* within an online learning scenario, RQ III.1 asks for the (general) suitability of *NN* within an online learning environment. In order to integrate *AutoML* systems into *online learning*, RQ III.2 refers to the first requirement defined by RQ III.3 goes one step further, assumes that an incremental adaptation of hyperparameters is possible and asks for the performance advances of incrementally adapted hyperparameters. Derived from these research questions, we formulate Hypothesis II by assuming that the incremental configuration of *AutoML* and *NAS* systems is possible and leads to better performances on data streams.

Hypothesis II (Stream Adaptation)
In an online learning environment, the incremental adaptation of hyperparameters enables superior performance on data streams by aligning the learning process with the online learning requirements defined by Bifet et al. [23].

In contrast to the research questions for Hypothesis I, RQ III integrates *AutoML* and *NAS* techniques into one main research question. This is, on the one hand, the result of the additional requirements for *online*

learning for both *AutoML* and *NAS* and, on the other hand, the application of similar optimisation techniques developed within the course of this work. Further, while Hypothesis I and the resulting research questions build on already existing *AutoML* and *NAS* systems, in RQ III the aim is to build a common *HPO* system that is applied to automatically configure *ML* pipelines and neural architectures in a streaming environment.

1.4 Contributions

The research towards adaptivity in *AutoML* has led to several contributions along with the hypotheses and research questions identified in Section 1.2. We divide the provided contributions into (i) *systems*, (ii) *frameworks* and (iii) *artifacts*. While the *systems* and *artefacts* provided in Contribution C I are collections of steps that enable the utility-based adaptation, the provided *frameworks* (Contributions C II and C III) follow the *Scikit-Learn*'s design principles [42, 185, 94]:

R III–1 All objects share a consistent and simple interface.

R III–2 All hyperparameters are directly accessible and exposed as public attributes.

R III–3 Algorithms are the only objects to be represented using custom classes. Data sets are represented as sparse matrices and hyperparameter names as well as their values are expressed as standard Python strings or numbers.

R III–4 Many *ML* tasks are expressible as sequences or combinations of transformations to data. Whenever feasible, algorithms are implemented and composed from existing building blocks.

R III–5 Whenever an operation requires a user-defined parameter, the library defines an appropriate default value.

These design principles avoid the proliferation of already developed framework code, aim to adopt simple conventions and thus limits the number of methods to be known to a minimum required for the execution of the proposed algorithms. Further, we provide various artefacts that were developed along with the systems developed in Contribution C I that allow an evaluation of the system against the related work. We conclude the main contributions of this work as follows:

C I. System towards utility adapted *AutoML* and *NAS*:
We provide a modular system that allows the adaptation of *AutoML* and *NAS* systems to an end-users utility. While the formalisation for a utility adapted *AutoML* and *NAS* framework resemble each other, their training, execution and thus utility preferences vary. The first system concerns the adaptation of *AutoML* and is exemplary evaluated based on *TPOT* [177] given different performance metrics. The second system concerns the adaptation of *NAS*. Since *NAS* systems require greater computational resources, we evaluated this system on the established *NATS-Bench* [67] data set given metrics that include preferences such as the *NN*'s latency.

This research has led to the development of several software artefacts. The proposed systems are implemented in *Python* and available as Software artefacts on *Github*[1]. To evaluate the utility-based *NAS* system against state-of-the-art multi-objective *NAS* approaches, we provide a *RL Open-AI Gym* [39] environment that builds on the *NATS-Bench* [67] data set and enables the execution of (multi-criteria) *RL* approaches to *NAS*. To evaluate our approach, we present a synthetic evaluation where we assume predefined utilities. Since a user study would lead to new research questions about the user interface and its experience to state preferences, we exclude an evaluation with user interfaces as it would blur the thread of this work.

C II. Online Deep Learning Framework:
In order to give answer to the general suitability of *NN* within an *online learning* scenario (RQ III.1), we provide a framework that combines the established frameworks *river* [173] for *online learning* and *PyTorch* for the development of *NN* algorithms in Python. Our framework follows the *Scikit-Learn* design principles and thus extends *river* by the broad capabilities of *PyTorch* for developing neural architectures. To orchestrate generated *NN* with other components from the *river* library, *online DL* follows the *river API*. Further, this framework enables the execution of *NAS* and further research in *DL* considering an *online learning* scenario. The aim of this framework goes beyond the evaluation presented in this thesis, as it fosters and unifies future research in the area of *online DL*.

C III. Online Automated Machine Learning Framework:
To enable *incremental HPO*, and thus the automated configuration of *ML* pipelines under the online assumption, we propose and provide *EvoAutoML*, an evolution-based online learning framework consisting of heterogeneous and connectable models that support large and diverse configurations spaces that adapts to the *online learning* scenario. The configuration space of this framework is variable in that it allows in combination with the *online DL* framework, the application of *NAS*. We refer within the evaluation for the automated configuration of *ML* pipelines identical to the frameworks name to *EvoAutoML* and for the configuration of neural architectures considering both *EvoAutoML* and *online DL* frameworks to *EvoNAS*. The configuration space of *EvoAutoML* is generated within components from the river [173] framework. *EvoAutoML* follows as *online DL* the *river API* to further extend its capabilities in orchestrating *ML* pipelines.

Both frameworks (C II-III) are inspired by the *Scikit-Learn* design principles and based on the *river* [173] framework. Instead of building *EvoNAS* in a separate framework, we combine *EvoAutoML* and *online DL* into one system to answer RQ III.3 for the application of *NAS* on data streams. However, as the course of this work will show, the complexity of *EvoNAS* requires further techniques, such as the use of network morphisms, to achieve comparable results. This lies in the nature of the underlying *online HPO* process of *EvoAutoML* and the nature of *NN* revealed by answering RQ III in the course of this work.

[1] https://github.com/kulbachcedric/, accessed on January 30, 2023

1.5 Outline

This thesis is structured into five parts that include the (i) overview, the (ii) preliminaries, the (iii) utility- and (iv) stream-based adaptation and the final part that concludes this work. The parts are thereby structured as follows and further illustrated in Figure 1.2:

Part II provides the preliminaries for this work. This part is split into two chapters: (i) the foundations that provide the main concepts building on the idea of automating the well researched *KDD* process and thus introducing and formalising the idea of *AutoML* and *NAS*; (ii) the related work that illustrates the allied research regarding the adaptation to a particular utility and the adaptation to data streams of *AutoML* and *NAS* systems. For the utility-based adaptation this includes related work in *metric learning* and *multi-objective ML* and for the adaptation to data streams the related work incorporates *online ensembles* as well as research carried out in the field of *online DL*.

Part III provides our approach towards the adaptation to a utility an end-user might pursue based on preference learning techniques (Contribution C I) and evaluates the proposed approach incorporating *AutoML* and *NAS* frameworks. This part of this thesis builds on the following publications:

- Kulbach,C., Philipp,P., and Thoma,S.,"Personalized Automated Machine Learning", ECML 2019.
- Kulbach,C., and Thoma,S.,"Personalized Neural Architecture Search", ICDMW 2021.

Part IV introduces the frameworks towards evolution-based online *AutoML* and *NN*. First, we present and formalise the algorithm of *EvoAutoML* and then give the methodology and the framework for *online NN* in detail. Within the framework description for *online NN* empirically evaluate the (general) suitability for *NN*'s (RQ III.1). Finally, we evaluate in this part the adaptability of *EvoAutoML* as well as the combination of the *EvoAutoML* and *online NN* to a *NAS* framework regarding related approaches. This part of this thesis builds on the following publication:

- Kulbach,C., Montiel,J., Bahri,M., Heyden,M. and Bifet,A. "Evolution-Based Online Automated Machine Learning", PAKDD 2022.

Part V concludes this thesis, provides an overview and outlook towards future research on utility-based adaption with and without consideration of feedback. Further, we depict the roadmap for the *EvoAutoML* and *online DL* frameworks developed within the course of this thesis.

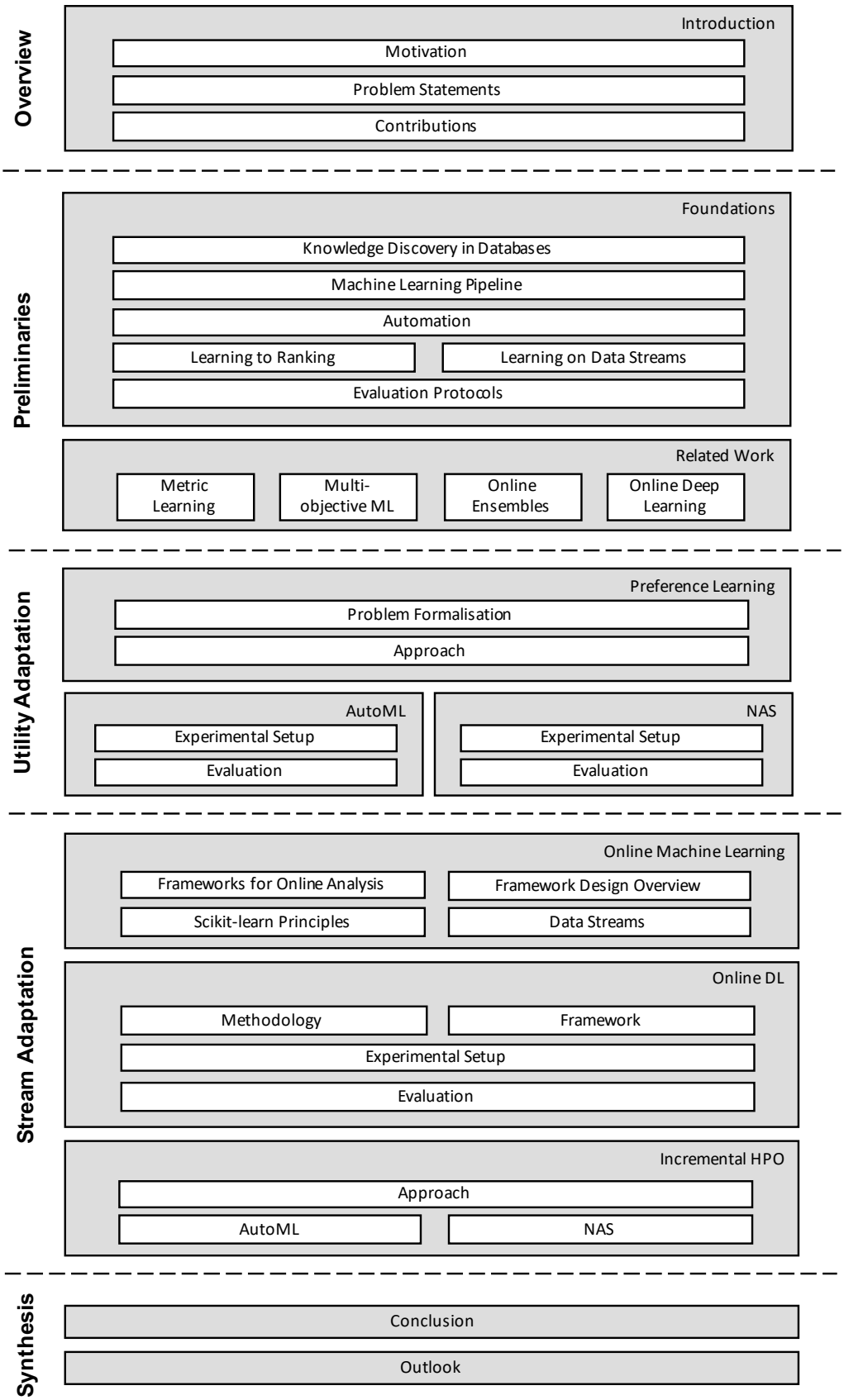


Figure 1.2: Structure and overview of this thesis

Part II

Preliminaries

In this part, the essential basics for this work are presented. Given the topic *"Adaptive Automated Machine Learning"* and starting from a Data Mining process, these are divided into ML pipelines, their automation, Neural Architecture Search and the foundations for an online learning environment.

2

Foundations

This chapter defines the foundations for this thesis. We approach *AutoML* from a data mining point by presenting in Section 2.1 the basic *KDD* process. Following this process, we define in Section 2.2 the *ML* Pipeline concept and introduce its principal components. To automate and optimise *ML* pipelines we discuss in Section 2.3 various *HPO* techniques as well as the concept of *ML* pipeline automation also referred to as *AutoML*. We introduce different ranking techniques in Section 2.4 that enable utility driven *AutoML* and *NAS* by learning the underlying utility. In Section 2.5, we switch from a supervised batch learning setting to a supervised incremental learning setting and discuss the essential requirements and concepts for incremental learning. To complete the foundations for this work, we present in Section 2.6 standard evaluation protocols for the batch and for the incremental setting as well as established metrics (Section 2.6.1) that are used within the evaluation protocols.

2.1 Knowledge Discovery in Database

KDD (*Knowledge Discovery in Databases*) encompasses the overall process of discovering useful knowledge from data [79] and thus assists humans in extracting useful information (knowledge) in an environment of exploding volumes of data. It seeks the establishment of standards in the field of *DM*, whereby it includes *DM* as one (key) component. From this point of view, the scope of a data scientist within the *KDD* process goes further than *DM*. Fayyad et al. [79] defines *KDD* as "the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data" [79], where a *non-trivial process* implies 5 steps (see. Figure 2.1). These steps involve the search process for retrieving useful patterns within the data. By *patterns* Fayyad et al. describe the structure of a set of data, where *data* is a set of facts. The patterns found are subject to the requirements that they must be (i) *valid*, (ii) *useful* and (iii) *understandable* for the system and the user [79]. When considering *AutoML* or *NAS* as an instrument that searches for patterns by optimising and applying *ML* algorithms, we investigate in this work the usefulness of found patterns by steering the optimisation process of *AutoML* system to a utility that goes beyond the predictive performance. And further, we investigate the validity of these patterns that occur continuously and may change over time within possibly infinite data streams by proposing an incremental optimisation technique for *ML* pipelines. In Figure 2.1, we depict the steps comprising the *KDD* process. A prerequisite for the process is to have or

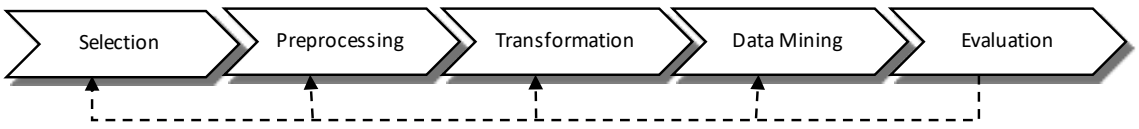


Figure 2.1: *KDD*

develop an understanding of the application domain and identify the process's goal. Within the first **Selection**

step, we create a target data set by selecting a data set or a subset on which the discovery is performed. After which, the **Preprocessing** step incorporates basic operations, such as handling missing values or removal of noise. In the next step, the **Transformation**, the goal is to find useful features that represent the data accordingly to the defined purpose of this process. Dimensionality reduction or transformation methods are applied to reduce the number of variables and thus lower the computation complexity or find invariant representations. The goal of the **Data Mining** step is to select appropriate methods that search for patterns in the data and thus match the overall criteria of this process. This includes deciding which models and parameters may be appropriate for searching patterns in the data. Lastly, the **Evaluation** step involves the interpretation of the mined patterns, where extracted patterns or the data passed to the selected models are visualised and thus evaluated. Building on the Evaluation step, it is possible to reflect on each of the previous steps and iterate over loops to match the process's goal or incorporate the gained knowledge into another system. In the following we broadly depict the *SEMMA* as well as the *CRISP-DM* processes as industry standards [6] and implementations of the *KDD* process.

2.1.1 SEMMA

The *SEMMA* (*Sample, Explore, Modify, Model, Assess*) process was developed by the SAS Institute, considers five steps and is designed to work with the Enterprise Miner software from the SAS institute. However, *SEMMA* is besides its implementation within the SAS Software a popular methodology for applying *DM*. In Figure 2.2 we depict this process, where one can see the similarity to the *KDD* process. It is likewise possible to reflect on each of the previous steps after the last step. The first step, **Sample**, of this process

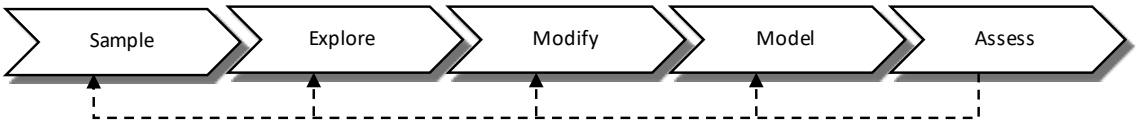


Figure 2.2: *SEMMA*

consists of sampling the data by extracting information that is big enough to contain important information and small enough to manipulate quickly. Within the **Explore** step, we get an understanding of the data by conducting univariate and multivariate analyses. This enables understanding each factor individually and finding relationships between the data collected. Based on the exploration, the data is similar to the transformation step in the *KDD* process parsed, cleaned and refined within the **Modify** step. In the **Model** step, we apply data mining techniques to produce a projected model of the final, desired outcome of the process. Within the last step, the **Assess**, the model is evaluated for its usefulness and reliability. As for the *KDD* process, it is possible to reflect on each of the previous steps and iterate over loops to match the process's goal.

2.1.2 Crisp-DM

Another popular process is proposed by Chapman et al. [50] that can guide the implementation of *DM* applications is *CRISP-DM* (*CRoss Industry Standard Process for Data Mining*). It comprises 6 steps that are iterated within a loop and visualized in Figure 2.3. The process starts with a **Business Understanding** step, where the focus lies on understanding the objectives and requirements from a business perspective. This knowledge is then converted into a *DM* problem. A preliminary process plan that achieves the objectives is defined in this step. Considering the goals of this work, the utility adaptation that enables to steer *AutoML* and *NAS* systems into a direction the end-user or domain expert pursues is located within this step. After

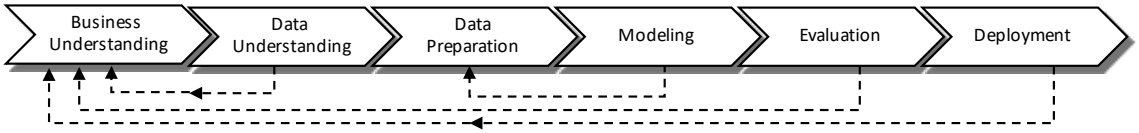


Figure 2.3: *CRISP-DM*

having developed the objectives and the requirements, the next step is the **Data Understanding**, where the goal is to get familiar with the collected data, identify data quality problems and detect subsets of information to develop initial hypotheses from the data. As visualised in Figure 2.3, there is a close link between the Business Understanding and the Data Understanding step; By formulating the *DM* problem, we need at least some understanding of the available data. The **Data Preparation** step comprises all activities to generate the data that will be fed into the chosen *DM* model. In this step, we further clean the data, select and generate functional attributes, as well as transform the data to then apply modelling techniques within the **Model** step. In this step, the search for suited modelling techniques and their parameters is carried out. Since some modelling techniques require specific data formats or one realises data problems while searching for suitable methods, the Modelling step is linked back to the Data Preparation step. Nowadays, this modelling step compromises a broad range of *ML* models At the **Evaluation** step, one or more models that appear to have high utility are evaluated to be sure that the previously executed steps achieve the objectives defined within the first step but also to determine if other types of malfunctions occur that were not yet been considered. If the model matches the requirements and the objectives, the model can be deployed within the final **Deployment** step. Depending on the requirements and making use of the created models, the Deployment step can reach from a simple report to a complex repeatable *DM* process.

2.1.3 Comparison

We presented the *KDD* process as well as the *SEMMA* and the *CRISP-DM* processes that define a set of sequential steps that pretend to guide the implementation of *DM* applications and thus to standardise the *KDD* process. Based on Azevedo and Santos [6], we want in this Section to broadly compare these processes to then transfer the correspondences into a learning process and thus enable an *AutoML* system. A summary of the correspondences of the different processes is presented in Table 2.1. Comparing the *KDD* and the

Table 2.1: Comparison of the correspondences between *KDD*, *SEMMA* and *CRISP-DM* [6]

KDD	***SEMMA***	***CRISP-DM***
Pre *KDD*	-	Business Understanding
Selection	Sample	Data Understanding
Preprocessing	Explore	
Transformation	Modify	Data Preparation
Data Mining	Model	Modelling
Evaluation	Assessment	Evaluation
Post *KDD*	-	Deployment

SEMMA process, the commonalities are striking in that the Selection step from the *KDD* can be identified with the Sample step from the *SEMMA* process, the Preprocessing step can be identified with the Explore step, the Transformation with the Modify step, the Data Mining with Model and the Evaluation step can be recognised with the Assessment step. However, since *SEMMA* is directly linked to the SAS Enterprise miner software, it can be seen as an implementation of the *KDD* process [6]. In Table 2.1, we defined

the initial phase and requirements of *KDD* as the Pre *KDD* step and the necessary steps to incorporate the gained knowledge into another System after the Evaluation step as Post *KDD* step. Comparing *CRISP-DM* with *KDD* the correspondences and boundaries of the steps are not as straightforward as for the *SEMMA* process. However, *CRISP-DM* incorporates a Business understanding step which can be identified with the requirements defined within the Pre *KDD* steps. Since the goal of the Data Understanding step is to gain an understanding of the collected data, identify possible data quality problems and develop initial hypotheses, the Data Understanding step comprises the Selection as well as the Preprocessing step from *KDD*. The Data Preparation step can then be aligned to the Transformation step, the Modelling to the Data Mining and the Evaluation step to the Evaluation of the *KDD* process. The Deployment step can be identified with the Post *KDD* step by utilising the developed models.

We presented the *KDD* process as one step towards a unified methodology for *KDD* and thus for *DM*. Furthermore, we depicted *SEMMA* and *CRISP-DM* as implementations of *KDD* and compared the processes by pointing out the commonalities within the different steps. As depicted within the introduction of this thesis in Figure 1.1, *AutoML* aims to automate the steps between the Selection and the Data Mining steps by building *ML* pipelines and performing *HPO*. However, it emerges that only parts of the *KDD* process are covered by the *KDD* process. In order to automate *KDD* and *DM* the Selection, Preprocessing, Transformation and Data Mining *AutoML* and *NAS* apply *HPO* techniques to optimize the process based on a predefined loss function. Within the utility-based adaptation of *AutoML*, we aim to investigate the business understanding step that defines an underlying objective and within the adaptation to data streams, we aim to investigate the adaptation to patterns that may change over time and thus, the evaluation step of the *KDD* process.

2.2 Machine Learning Pipeline

Following *KDD*, we define in this section the concept of a *ML* pipeline considering (i) Data Preparation (Section 2.2.2), (ii) Feature Preprocessing (Section 2.2.3) and (iii) Supervised Learning Models (Section 2.2.1). In the latter, we present, besides the foundations for various supervised learning models, also *NN*s like a foundation for *NAS*, as well as *ensemble* models.

Before introducing the concept of a *ML* pipeline and its components, we highlight the commonalities and differences between *KDD* and *ML*. Since both *KDD* and *ML* fall under the aegis of Data Science, and both are valuable tools for solving complex problems, the lines between these terminologies become blurred. While *KDD* is about techniques and tools used to unfold patterns in data that were previously unknown and make data more usable for analysis, *ML* aims at training machines based on gathered data to perform complex tasks that incorporate tasks, such as retrieving patterns in a supervised or unsupervised manner. Here, the differences become apparent. While *KDD* relies on the overall process of discovering valuable and understandable knowledge, *ML* is concerned with training models that perform a specific task [131]. This significant difference incorporates that *KDD* relies on human intervention and is created for use by people. These differences, however, allow them to complement each other, as *ML* techniques are handy tools in *DM* to discover regularities that can be found automatically. For instance, in *ML* there is an effort to integrate humans (*Human Guided Machine Learning*) within the learning process and in *KDD ML* is a powerful instrument that is nowadays indispensable. Our approach aims to steer *AutoML* systems to a particular utility and thus aims to adapt *AutoML* to obtain useful *ML* pipelines.

ML is the process of discovering algorithms that have improved courtesy of experience derived from data. It's the design, study and development of algorithms that permit machines to learn without human intervention. However, *ML* algorithms can be divided into supervised and unsupervised learning techniques based on the data structure, but also into online and offline learning algorithms based on the availability of data. In

Section 2.2.1, we formalise supervised offline *ML* as well as the concept of *ML* pipelines. The distinction to *online learning* is made within Section 2.5.

2.2.1 Supervised Machine Learning

In supervised *ML*, one assumes each instance of a data set can be assigned to classes or numerical values for predicting diverse outcomes. This can also be referred to as labelled data. If one assumes that all data are available simultaneously, one speaks of *offline* or *batch learning*. In Definition 1, we formalise the supervised offline learning problem.

➢ **Definition 1.** Supervised offline learning
Let $\mathcal{D}$ be a set of data points $\mathcal{D} = \{(\overrightarrow{x}_1, y_1), \ldots, (\overrightarrow{x}_t, y_t)\}$, then the task is to learn a function $f^* : X \to Y$ (also referred to as model), that transforms a feature vector $\overrightarrow{x} \in X$ into a target value $y \in Y$ using all data points at once. Furthermore, let $\mathcal{L}(f(\overrightarrow{x}), y)$ be a loss function that quantifies the correctness of a series of predictions of f trained on $\mathcal{D}_{train} \subset \mathcal{D}$ for $\overrightarrow{x}$. Denote that $\{\overrightarrow{x}, y\} \notin \mathcal{D}_{train}$ and $\mathcal{D}_{train} \cap \mathcal{D}_{valid} = \emptyset$. Then the goal to is to minimise the loss $\mathcal{L}$ on $\mathcal{D}_{valid}$ using a validation protocol $\mathcal{V}(\cdot, \cdot, \cdot, \cdot)$.

$$f^* \in \min \mathcal{V}(\mathcal{L}, f, \mathcal{D}_{train}, \mathcal{D}_{valid}) \tag{2.1}$$

Assuming that Y is a set of categorical values, Definition 1 is referred to as classification problem and when $Y \in \mathbb{R}^n$ Definition 1 is referred to as a regression problem. In addition to the supervised offline learning problem, we also added in Definition 1 the notion of a function or model f, that transforms a feature vector into a target value based on the given data set. Most *ML* models have settings also denoted hyperparameters that are set before the model learns based on the given data set. These parameters are set in dependence on the data and are crucial for the model's performance. To influence the external settings of a model, we assume that each algorithm has parameters $\lambda \in \Lambda$ that can be set in advance of the training process. We define a hyperparameter as follows:

➢ **Definition 2.** Hyperparameter
Let $\mathcal{A} = \{A^{(1)}, \ldots, A^{(n)}\}$ be a set of algorithms, then each algorithm $A^{(i)} \in \mathcal{A}$ is configured by a vector of hyperparameters $\overrightarrow{\lambda}^{(i)} \in \Lambda_{A^{(i)}}$. Whereby, $\Lambda_{A^{(i)}}$ denotes the configuration space of $A^{(i)}$, e.g. learning method, optimiser or thresholds for prediction.

The hyperparameters of a model f can range from the underlying optimiser, complexity thresholds, e.g., memory consumption, or similarity measurements to approximate the distance between data points. These parameters have a high impact on the model's performance and are optimised within the modelling step when referring to the *KDD* process. We have defined the supervised offline learning setting in Definition 1, where we assume that a function $f : X \to Y$ transforms a feature vector into a target value. Furthermore, we introduced hyperparameters where we assume that each algorithm $A^{(i)} \in \mathcal{A}$ can be externally configured in advance of a learning process by its hyperparameters $\overrightarrow{\lambda}^{(i)}$ from its domain $\Lambda_{A^{(i)}}$. Based on the Definitions 1 and 2, we can now define a *ML* pipeline that is capable of transforming a feature vector $\overrightarrow{x}$ into a target value y as does a function or model f. By a set $\mathcal{A}$ of defined algorithms and their configurations space Λ_A we

can concatenate these algorithms within a pipeline $\mathcal{P}$ that represents the function $f : X \rightarrow Y$. We assume that the concatenation of algorithms can be modelled by a *DAG*, where each node represents an algorithm, and the edges represent the flow of the input data through the different algorithms. The *DAG* restricts the concatenation of algorithms in that no data is passed back within the *ML* pipeline. Denote that this structure is likely to be more restricted in applied *ML* pipelines, e.g. the last step of the pipeline is an algorithm that finally performs the classification or regression step in supervised learning. Accordingly, to the definitions of a model and its hyperparameters, we define a *ML* pipeline as follows:

➢ **Definition 3.** *ML* Pipeline [253]
Let a triplet $(g, \overrightarrow{A}, \overrightarrow{\lambda})$ define an *ML* pipeline with $g \in G$ a valid pipeline structure, $\overrightarrow{A} \in \mathcal{A}^{|g|}$ a vector of the selected algorithm for each node and $\overrightarrow{\lambda}$ a vector comprising the hyperparameters of al selected algorithms. The pipeline is denoted as $\mathcal{P}_{g,\overrightarrow{A},\overrightarrow{\lambda}}$. The learning goal of a configured pipeline $\mathcal{P}_{g,\overrightarrow{A},\overrightarrow{\lambda}}$ can be defined accordingly to Definition 1 as follows:

$$\mathcal{P}^*_{g,\overrightarrow{A},\overrightarrow{\lambda}} \in \min \mathcal{V}(\mathcal{L}, \mathcal{P}_{g,\overrightarrow{A},\overrightarrow{\lambda}}, \mathcal{D}_{train}, \mathcal{D}_{valid}) \tag{2.2}$$

Following the *KDD* process domain experts, but also the best common practices [253] commonly build a *ML* pipeline considering (i) data preparation (also referred as data cleaning), (ii) feature engineering and (iii) modelling algorithm (see Figure 2.4). At first, the data is cleaned by searching for errors in $\mathcal{D}$ and then repairing them. Within the feature engineering step new features are created and relevant features are selected. Within the final step based on the generated and selected features a suited model is selected and trained. This prototypical *ML* pipeline is usually adapted and extend by data scientists. In the following, we

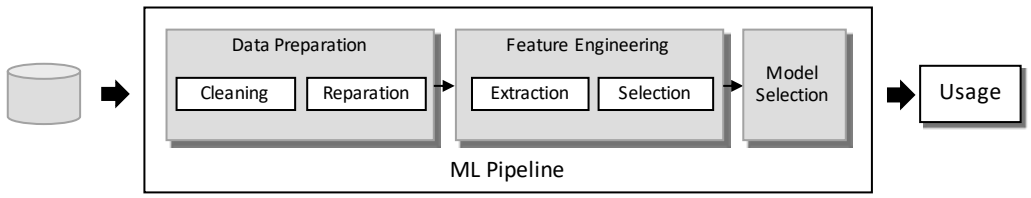


Figure 2.4: Illustration of a *ML* pipeline consisting of (i) data preparation, (ii) feature engineering, and (iii) model selection.

present the typical steps of the *ML* pipeline as depicted in Figure 2.4. Algorithms for the data preparation step will be depicted in Section 2.2.2, algorithms for the feature preprocessing step in Section 2.2.3 and in Section 2.2.1 we present common models that can be chosen within the model selection step.

2.2.2 Data Preparation

The purpose of data preparation is to improve the quality of the given data by i.ex. Removing data errors, thus, has a strong influence on the later steps of the *ML* pipeline, as the preparation of data makes the further steps possible. Its process can be split accordingly to Chu et al. [55] into a (i) error detection and an (ii) error repairing task. Further, it is mainly applied based on predefined rules executed when i.ex. an error has been detected.

2.2.2.1 Error Detection

Errors within the data at hand can be classified into missing values, redundant entries, invalid data formats or broken links between entries when merging multiple data sources [195]. For detecting errors, most techniques [96, 235, 244] involve humans that decide whether a value is an error or not, e.g. to identify if an instance corresponds to a duplicate and thus is redundant or not. Some approaches [30, 55, 49] detect errors automatically by evaluating predefined functional dependencies or learning separate *ML* models (e.g. in anomaly detection). Commonly *AutoML* systems apply simple error detection mechanisms, such as removing or repairing instances that are not a number within a feature that contains only numbers.

2.2.2.2 Error Reparation

The subsequent repairing task is usually carried out automatically, e.g. by modifying the data set concerning minimal changes or reparation specified by rules. The imputation of missing values, i.ex. it can be applied with different strategies, such as using the mean, median, the most frequent value from other instances of the feature, or predefined imputation values. Redundant entries, such as duplicates, can be removed from the data set, and invalid data formats can be reformatted based on rules. However, most data cleaning algorithms are rule-based and hardcoded. This step influences the performance of the underlying *ML* model in that it allows its execution without errors that might result in misleading predictions. Further, as *SVM*'s require numerical encodings of categorical features while tree-based approaches (see Section 2.2.4) allow categorical values, a suitable error reparation technique enables the execution of further *ML* pipelines. Referring to Definition 3, the applied cleaning method within the *ML* pipeline can be seen as a node of the *DAG* g and thus as an algorithm $A_{\lambda}^{(clean)}$ configured by λ.

2.2.3 Feature Preprocessing

While data preparation techniques primarily enable the execution of *ML* pipelines, feature engineering has a significant impact on the pipeline's predictive performance. It can be seen as the process of generating and selecting features from a given data set for the subsequent modelling step [253]. Thus, it summarises all types of preprocessing steps [94] that go towards (i) feature selection and (ii) feature extraction (see Figure 2.4). The feature preprocessing step is highly domain-specific and difficult to generalise. For example, when considering a categorical feature that is encoded within the data set $\mathcal{D}$ as a numerical value, the underlying and subsequent algorithm would treat this feature as a numerical feature, where a higher or lower value does not reflect the category. Even for data scientists, assessing a feature's impact is difficult, as domain knowledge is necessary, e.g. to encode categorical features. A commonly applied technique is to calculate the covariance matrix to measure the feature importance and the correlation between numerical features. This step enables the combination of features that point into the same direction. In the following we first formally define the feature preprocessing step and then depict the different types in more detail.

➢ **Definition 4.** Feature Preprocessing, adapted from [127]
Let $\mathcal{D} = \{(\overrightarrow{x}_1, y_1), \ldots, (\overrightarrow{x}_t, y_t)\}$ be a set of data points, then the task is to learn a feature representation $\phi : X \rightarrow F$ that maps the input vectors $\overrightarrow{x}$ to feature vectors $\phi \in F$. We want to infer a function f (see Definition 1) with the aim that its metric loss $\mathcal{L}(f(\phi(\overrightarrow{x}), y)$ is lower than $\mathcal{L}(f(\overrightarrow{x}), y)$.

The search for a suitable representation ϕ that increases the performance of f requires notably domain knowledge and also shows in Definition 4 a close connection and dependence to the learned function f. Thus, it can similar to the data preparation step be seen as a node $A_{\lambda}^{(preprocessing)}$ of the *DAG* g and thus be integrated into a *ML* pipeline (see Definition 3). Also, the concatenation of multiple preprocessing steps might be considered within a *ML* pipeline. Furthermore, Definition 4 shows the possibility of automating this step by testing various preparation techniques ϕ that lead to better performances regarding the metric $\mathcal{L}$. As for the data preparation step in Section 2.2.2, feature preprocessing can be divided into (i) feature extraction and (ii) feature selection, which are depicted in more detail in the following. While feature extraction aims to generate new features or transform existing features that minimise the metric of $\mathcal{L}(f\phi(\overrightarrow{x}), y)$, feature selection aims to compress the given features into a smaller feature representation by removing non-influential and selecting useful features.

2.2.3.1 Feature Extraction

The feature extraction step generates new features by combining existing features to produce more useful ones. However, a common approach for extracting new feature values is to apply a set of predefined operators to the original feature values. These operators can be grouped into unary, binary and high dimensional operators. Unary operators extract from a single original feature value new feature values. For instance, if the original data set incorporates dates, one can extract additional features such as the year, month, weekday or whether the given date was on the weekend describing seasonal influences. Binary operators combine two feature values by applying basic operations (e.g. $+, -, *$) or comparing these feature values using correlation tests and regression models [253, 127]. The choice of operators is based on domain knowledge and the type of the underlying feature values. Operators with a higher-order use multiple feature values to create one or several new features. For example, several features' average or mean values can be used as a new feature. However, to extract new features by applying predefined operators, one can also use algorithms that automatically learn informative features and reduce feature preprocessing costs compared to manual approaches. The *PCA* is one technique commonly used for feature extraction in high-dimensional feature spaces [200] and also integrated into *Auto-Sklearn* (see Section 2.3.2). Applying various kernels allows for extracting linear and non-linear components and, thus, extracting (automatically) helpful features. *NN*s have shown their ability to extract valuable features by applying, e.g. auto-encoders that learn a feature representation through a bottleneck layer of the network's architecture. Thus, they implicitly apply representation learning, where the task is to learn how to represent features using simple computational units. The extracted features are represented by the bottleneck of the auto-encoder and thus a dense representation that implicitly combines and selects suitable features.

2.2.3.2 Feature Selection

Feature selection is the task of selecting the most valuable features from a given data set to improve the *ML* pipeline performance. By selecting the most valuable features and removing redundant or misleading features, less data is passed into the pipeline, and thus costly computations are reduced. Furthermore, by bypassing less features into the *ML* pipeline, the interpretability for predictions made are improved as less features are considered within the underlying *ML* model. It might enhance the performance of the pipeline [203] as redundant noise is removed from the data set used for training. The main feature selection techniques can by characterised by their interaction with the function f (see Definition 4) into (i) filter, (ii) wrapper and (iii) embedded techniques [203].

Filter techniques assess the use of the features by looking only at the intrinsic properties of the feature values that can be divided into univariate and multivariate methods and are independent of the *ML* model. Univariate filter techniques consider each feature separately and ignore feature dependencies e.g. χ^2-test or information gain [17]. Multivariate filter techniques consider feature dependencies e.g. by selecting features based on the correlation [105]. Further examples for multivariate filter techniques that do not incorporate a selected *ML* prediction model f are the Markov blanket filter [132] or the fast correlation based [248] feature selection.

Wrapper techniques embed *ML* models within the feature selection step. Subsets of possible features define a search space that is evaluated in conjunction with the model on a loss metric $\mathcal{L}$. Therefore, the search for a specific subset of features is performed by training a model on the subset and testing it based on $\mathcal{L}$. The search for valuable features is thus wrapped around the model and can be performed automatically. Search techniques can be grouped into deterministic and heuristic methods [254]. Search heuristics (*HPO*) that can be applied not only on *AutoML* but also on feature selection tasks are depicted in more detail in Section 2.3. Siedlecki and Sklansky [212] compare different approaches such as branch and bound optimization techniques that lead to optimal solutions and state that due to the exponential growth of possible feature combinations the application of heuristics such as simulated annealing [216] and genetic algorithms [125, 152, 179] (see Section 2.3) is more suitable for an automated and wrapped feature selection than the application of approaches that guarantee an optimal solution.

When the search for suitable features is integrated into the model, embedded techniques are applied. For instance weighted random forest [63, 231] selects features within an ensemble of classification trees (see Section 2.2.4.1) that are used as decision nodes. But also *NN*s are successfully applied [12, 165] to select suitable features and to perform a supervised learning task. The wrapper and the embedded techniques can be combined with feature extraction but are mapped to a specific model.

For instance, by learning weights of *NN*s, features can be extracted as well as selected. This also shows that the boundaries between the individual steps displayed in Figure 2.4 can sometimes not be distinguished, but also that the data preparation, as well as the feature engineering step, can be automated in connection with the underlying model. These models can either be predictive models used in the latter step of the *ML* pipeline to evaluate a feature set [212, 216, 125, 152, 179] or a model used to perform *HPO* [63, 231, 103]. To complete the *ML* pipeline depicted in Figure 2.4 and to showcase the main differences in concepts that are suitable to solve the supervised learning problem (see Definition 1), we present in the following the main learning techniques.

2.2.4 Learning Models

In this section we present the foundations for different algorithms that are capable to minimise the metric $\mathcal{L}$ using a validation protocol $\mathcal{V}(\cdot,\cdot,\cdot,\cdot)$ (see Definition 1). The most common approaches to solve Equation 2.1 are decision trees, *SVM* (*Support Vector Machine*), Bayes based, neighbourhood based and *NN* (*Neural Network*) approaches. By highlighting the differences, strengths and weaknesses of these concepts, we showcase the heterogeneity and thus the importance of the model selection step. Furthermore, the selection of the underlying model influences the needed data preparation steps in that some models are constrained in their feature values. For a more profound insight into different *ML* concepts we refer to [94, 101, 170].

2.2.4.1 Decision Tree

A decision tree is first and foremost a method for approximating discrete-valued targets $y \in Y$ (classification) that uses a tree like model for its decisions. They learn simple decision rules inferred from the instances $\overrightarrow{x} \in X$ [192] by sorting them down in a tree from the root to some leaf node. The last leaf node provides the classification of the instance $\overrightarrow{x}$. Each node of the tree represents a tested feature and has two or more branches that constitute values for the tested attribute. Quinlan [192] propose a greedy algorithm that grows the tree from the root to its leafs top-down. The algorithm selects the attribute that best classifies the used training examples at each node. The choice of a feature's worth for a decision node is commonly measured by the information gain, which measures how well a given feature separates the training examples according to the target y. This process continues until the tree perfectly classifies the training examples $(\overrightarrow{x}, y) \in \mathcal{D}_{train}$, or until all features have been used. However, besides this greedy heuristic, there have been several extensions and improvements [193] for decision tree learners, and the advantages are evident. Decision trees can easily be visualised and thus are simple to understand and interpret. Furthermore, due to the structure of the tree, the computational costs for predicting $\hat{y}$ are logarithmic in the number of instances used to train the tree. Disadvantages appear when learners that build a decision tree create over-complex trees that do not generalise on the data set $\mathcal{D}$. This effect often appears when learning regression tasks since the label y can have an infinite number of expressions. To avoid over-complex decision trees is to limit the tree size or its memory consumption.

2.2.4.2 Support Vector Machine

Another powerful and versatile concept for learning supervised problems are *SVM*'s [32, 57]. They construct a hyperplane, or set of hyperplanes in a high dimensional space that maximises the margin between the training features $\overrightarrow{x}$ and the hyperplanes. Intuitively, a good separation is achieved by the hyperplane with the most significant distance to any class's nearest training data points. In general, the larger the margin, the lower the generalisation error of the classifier. The planes used to build the hyperplane and measure the distance passing through a class's points are called support vectors. The learned hyper-planes are then used to predict based on the feature vector $\overrightarrow{x}$ the value $\hat{y}$. If all points are separable by a hyperplane, this can be defined by a hard margin *SVM*. However, this is not the default due to data contamination. To solve this problem, a soft margin *SVM* uses a tolerance to violations of the hyperplane employing a hyperparameter. One key innovation associated with *SVM* is the kernel trick. The kernel trick observes that many machine learning algorithms can be written exclusively about dot products between examples. It enables a non-linear classification and is mainly implemented as a further hyperparameter. However, to extend *SVM* for regression tasks, the objective to find a hyperplane that splits the classes of $\mathcal{D}$ is reversed in that the goal is to find a hyperplane and support vectors that fit as many instances as possible between the support vectors and thus to predict a regression label $\hat{y}$. Since *SVM* models use only a small subset of training points to create the support vectors, they have the advantage of being memory efficient. Furthermore, by choosing a soft margin *SVM*, the classification function can be adapted to inconstant data and thus to the data preparation step presented in Section 2.2.2. A disadvantage of *SVM* is that if the number of features is much higher than the number of samples available within $\mathcal{D}_{train}$, avoiding over-fitting in choosing Kernel functions and regularisation terms, are crucial and require deep domain and model knowledge.

2.2.4.3 Neighbourhood

Algorithms based on the search in the neighbourhood of a feature vector $\overrightarrow{x}$ are usually used in unsupervised learning settings. In the unsupervised scenario, the task is to cluster data points based on the distance

measurement of $\overrightarrow{x}$. Fix and Hodges [83] propose with *k-NN* (*k-Nearest Neighbors*) an approach that classifies data points based on a majority voting of the k closest already seen data points ($\mathcal{D}_{train}$). Thus, the training of the algorithms consists of storing the labelled data points to estimate the closest relation within the feature space X on unseen data points. With the given labelled data points, *k-NN* can predict an unlabelled data point $\overrightarrow{x}$ based on the distance measure. Commonly used distance measurements are e.g. *Minkowski*, *Manhattan* or a simple *Euclidean* distance. To perform a regression task, each of the k nearest data points contributes uniformly or based on their weighted distance to the regression estimation of a feature vector $\overrightarrow{x}$.

However, with an increasing amount of data points and with a high number of features, *k-NN* gets computationally expensive. To avoid a computationally expensive distance estimation, the dimension of $\overrightarrow{x}$ can be reduced by dimensionality reduction techniques (see Section 2.2.3). Also, not all data points are necessary to be stored for a class estimation of an unlabelled data point. Data reduction techniques remove unnecessary data points from the training data set, reducing the computation complexity. Besides the *k-NN* approach where the k nearest neighbours provide the distance estimation and thus the label y, a radius based approach can be applied, where a majority voting is performed based on all data points within a radius r. Besides the classification, regression tasks can be performed by averaging the label y of the k nearest neighbours or the instances within the radius r.

2.2.4.4 Naïve Bayes

NB (*Naïve Bayes*) is a simple method to perform classification tasks. They are based on applying the Bayes' theorem with the assumption of conditional independence between the features $\overrightarrow{x}$ given the label y. For example, an apple could be classified as an apple (label y) if it is red, round and has a diameter of about 10 cm (features $\overrightarrow{x}$). The assumption in *NB* considers each of these features to contribute independently to the probability that the apple is classified as an apple, regardless of possible correlations between the colour, roundness, and diameter features [249]. The Bayes' theorem is defined by

$$P(y|x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n|y)}{P(x_1, \ldots, x_n)} \tag{2.3}$$

and enables the estimation of the conditional probability $P(y|x_1, \ldots, x_n)$ for a label y given a feature vector $\overrightarrow{x}$. The probabilities $P(x_1, \ldots, x_n)$ and $P(y)$ are retrieved from the data set and are constant given the input and by assuming that all features of $\overrightarrow{x}$ are independent

$$P(x_i|y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y) \tag{2.4}$$

the probability $P(y, x_1, \ldots, x_n)$ for a label y given the feature $\overrightarrow{x}$ can be estimated by

$$P(y, x_1, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, \ldots, x_n)}. \tag{2.5}$$

The label $\hat{y}$ can be estimated based on the maximal probability, whereby $P(x_1, \ldots, x_n)$ can be negotiated, as it is constant given an input $\overrightarrow{x}$:

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i|y) \tag{2.6}$$

It is clear that the assumption of conditional independence is a strong limitation for real-world applications and has led to various extensions to the *NB* approach, e.g. *GNB* (*Gaussian Naive Bayes*) [199], augmented

NB Zhang [249] or multinomial *NB* to support various data distributions. However, Caruana and Niculescu-Mizil [48] showed in a comprehensive comparison that approaches based on Bayes are outperformed by other classification algorithms, such as boosted trees or random forests but provided in few cases superior performances. A further disadvantage of this approach is the limitation to classification tasks.

2.2.4.5 Neural Networks

Neural networks, in general, can range from simple to complex, multi-layered structures. Due to their versatile *NN* topologies, they can be used in a wide variety of applications such as image and natural language processing [109] and showed impressive performances not only in supervised learning tasks. They are commonly employed in *RL*, unsupervised learning (auto-encoders) and time series forecasting tasks. Because of their importance for *NAS* algorithms, *NN* are discussed in more detail in this subchapter. In the literature, *NN*'s are often associated with the modelling of the human brain due to their artificial neurons and type of communication along weighted channels. The history of neural networks has been awe-inspiring in recent years, with its origins in the work of McCulloch and Pitts [162]. To further illustrate the parametrisation space and the variety of *NN*'s as well as the importance of their architectures, we introduce *NN*'s by modelling a simple logistic regression model as a *NN* and then depict more complex *NN* architectures that finally provided the breakthrough impact. We furthermore formalise *NN*'s for their application in *NAS*.

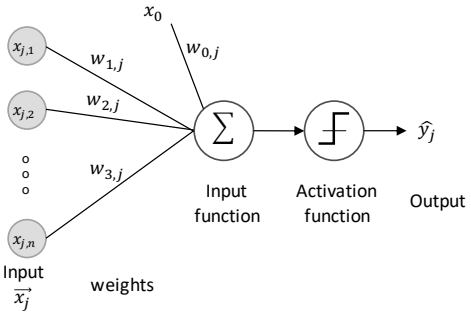


Figure 2.5: A mathematical model for a neuron adopted from Russell et al. [202].

The elementary component of a *NN* is a single artificial neuron depicted in Figure 2.5. As the figure shows, a neuron consists of an input $\overrightarrow{x_j}$, where each element $x_{j,1}, \ldots, x_{j,n}$ is weighted by $w_{j,i}$, including a bias $x_0 = 1$ weighted by $w_{j,0}$ within an input function [202]:

$$in_j(\overrightarrow{x_j}) = \sum_{i=0}^{n} w_{i,j} * x_{j,i} \tag{2.7}$$

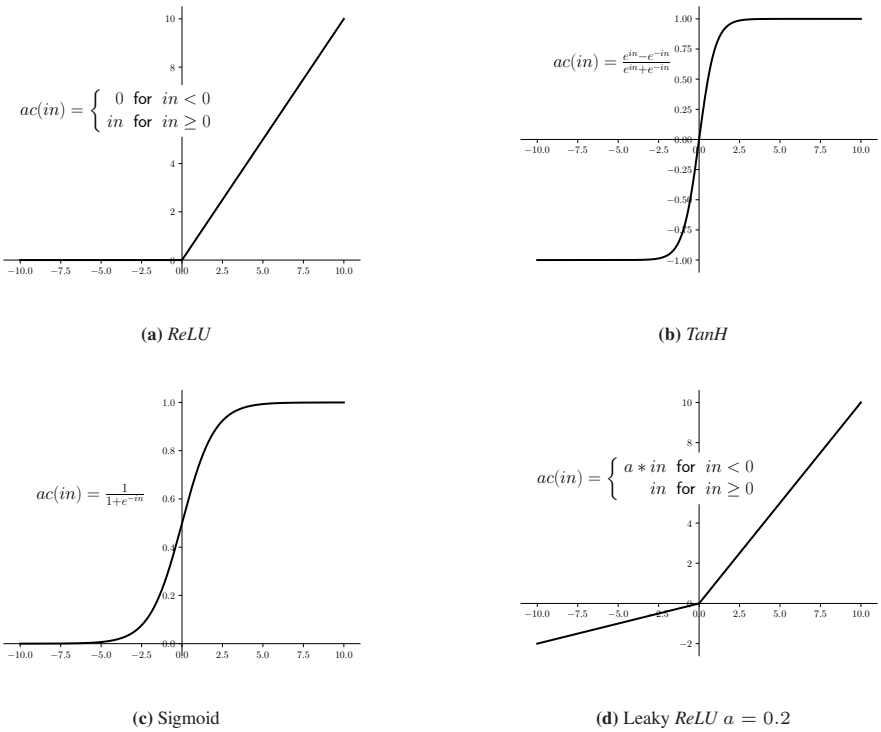
An activation function in is applied to this input function to calculate the neuron's output based on the weighted input. Most neural networks exhibit a layered or clustered structure, where multiple neurons are combined into one layer and then concatenated within the *NN*'s architecture. Whereby the first layer is called the *Input Layer* and the last layer *Output Layer*. If these layers are connected only in one direction, without loops, they are called *FNN* (*Feed Forward Neural Network*). *MLP*'s are a prominent example for *FNN*'s where multiple layers are concatenated, and all layers are fully connected.

The activation function of a neuron (see Figure 2.5) are of great importance for the modelling of *NN*'s because they enable us to learn complex functions as, without them, linear functions are generated as outputs, which

limits the ability to learn complex data. Thus, the output of the activation function can be used as input for further layers or as prediction output:

$$\hat{y} = ac(in_j) = ac(\sum_{i=0}^{n} w_{i,j} * x_{j,i}) \tag{2.8}$$

By applying non-linear activation functions, it is possible to map non-linearity between inputs and outputs, allowing more complex problems to be solved. In the modelling of artificial neurons, a distinction is made between different types of activation functions. The most commonly applied activation functions are *ReLU*, Leaky *ReLU*, *TanH*, Sigmoid and Softmax. The application of the activation functions depicted in Figure 2.6



(a) *ReLU*

(b) *TanH*

(c) Sigmoid

(d) Leaky *ReLU* $a = 0.2$

Figure 2.6: Commonly used Activation Functions: (a) *ReLU*, (b) *TanH*, (c) *Sigmoid*, (d) *Leaky ReLU*

can be developed based on their advantages and disadvantages and thus is highly dependent on the task and the domain knowledge. *ReLU*, for example, has the advantage of being very resource-efficient, as the output is always greater than 0 but does not bound the result of the output to higher values. In contrast, Leaky *ReLU* differentiates negative input values. The Softmax activation function is generally used as a categorical activation function as it compromises all inputs. It compresses the outputs in that they range between $(0, 1)$ by dividing through the inputs of other neurons within the layer:

$$ac_i(\overrightarrow{in}) = \frac{e^{in_i}}{\sum_{j=1}^{K} e^{in_j}} \tag{2.9}$$

The Softmax activation function depicted in Equation 2.9 can thus be interpreted as a probability for a class, whereby the sum of the results equals 1. However, the architecture is already crucial for the predictive performance and highly depends on domain knowledge. For instance, considering a neural architecture with only one output neuron and a Softmax activation function, the output remains constant regardless of

the input. *TanH* is a logistic function that maps the outputs to the range of $(-1, 1)$. It can be used in binary classification and produces two classes accordingly within the range $[-1, 1]$. The Sigmoid activation function is another logistic function that normalises the output of a neuron to $(0, 1)$. As it normalises the output and is independent of other neurons within the layer, it can be used for binary classification tasks.

The most straightforward neural architecture commonly applied in regression tasks is the linear regression and for classification tasks the logistic regression. By summarising the weighted bias $b = w_{0,j} + x_0$, we receive the prediction function for a linear regression model, where $w_{i,j}$ are the regression coefficients and b denotes the intercept:

$$in_j(\overrightarrow{x_j}) = \sum_{i=1}^{n} w_{i,j} * x_{i,j} + b \tag{2.10}$$

By passing this input into a Sigmoid activation (Figure 2.6c) function, we receive a (binary) Logistic Regression model, that is broadly used in statistics as well as in *ML*:

$$P(Y = 1|X = x_j) = \frac{1}{1 + e^{-x_j^T w_j}} \tag{2.11}$$

This means, we can think of Logistic Regression as a one-layer *NN*. Further, seeing the linear and logistic regression models as the simplest architecture of a *NN* that are already broadly applied in *online* and *offline learning* indicates the suitability of *NN*'s in *online learning* (RQ III.1), which will be useful in the course of this thesis and particularly for Chapter 8 , where we present the the *online DL* framework.

However, besides the activation function, the concatenation of different layers, as well as the input function (see Figure 2.5), shows the variability of possible neural architectures. In the context of *NN* with multiple (hidden) layers, the term *DL* and *DNN* is often used. Besides a *MLP* another well-known form of *NN* are *RNN*, where the neurons can have connections travelling in both directions by introducing loops into the *NN* architecture. This means that a neuron is also influenced by its own but temporally previous activation. The modelling is, therefore, much more dynamic than in the case of *FNN* and takes temporal behaviour patterns into account. However, *RNN*s bring challenges that need to be overcome, such as the fact that the gradient required to train the weights of the *NN* is challenging to obtain, as it needs to be propagated, not only over several layers but also back over time [174]. *LSTM* (*Long-Short Term Memory*) offer a solution to stabilise this gradient problem. In addition to the traditional *RNN*, these neural networks contain memory blocks in the recurrent hidden layers and thus can be trained similar to *FNN*s. On the one hand, these blocks consist of memory cells that store the temporal state of the network via self-connections. Furthermore, they contain gates that enable the flow of information to be controlled [111]. Other well-known *RNN*s are Hopfield networks and Boltzmann machines [202].

Besides, the backward connections of *RNN*s architectural changes of the layers have led to the outstanding performances of *NN*s on various learning tasks. One of the most popular types of *DNN*s are *CNN* (*Convolutional Neural Network*), which were introduced by LeCun et al. [146] in 1989. They gained success in recent years particularly in image classification [138, 145], *NLP* (*Natural Language Processing*) and computer vision [4]. They mainly involve sliding a filter over the layer's input and thus gathering the activation of the filters, which makes them particularly good at processing data that can be organised into a grid structure. This applies, for example, to time series that can be arranged in the form of one-dimensional grids or two-dimensional pixel grids that are present in image data. A *CNN* layer can be parametrised by its convolution as well as a pooling operation that further reduces the dimensions of the filter's output. *CNN* architectures have shown impressive results on the most common data sets for image classification.

A breakthrough to state-of-the-art in image classification was achieved in 2012 with the developed *AlexNet* [138] architecture, which obtained a then remarkable performance on image classification competitions. By

increasing the number of convolutional layers Simonyan and Zisserman [215] proposed in 2015 *VGGNet* that outperformed *AlexNet*, but also increased the number of trainable parameters (weights) from 62.4 million to 138.3 million parameters. However, the increasing number of convolutional layers leads to an impediment to the convergence of the model during training [107, 18]. This problem is also called vanishing or exploding gradients. In 2016 He et al., introduced instead of increasing the number of convolutional layers, the concept of residual learning, where skip connections are used to bypass some layers [107]. Another concept for improving *CNN*'s is *InceptionNet* [225], where the structure of the *CNN* layers are twisted by using filters of various sizes in one layer of the neural architecture. With an ever-increasing availability of computing power, Google introduced the search for the best *CNN* architecture with *NASNet* [255, 256] and thus paved the search for suitable neural architectures (*NAS*). In particular, research on *CNN*'s has created a remarkable change in terms of ways to solve complex problems in a wide variety of domains. We further discuss the development for *NAS* developments in Section 2.3.3.

We discussed the basic concepts, as well as the constituents of *NN*s; and to perform *NAS* within the course of this thesis, we formalise a *NN* in Definition 5. There have already been numerous formalisations [13, 82] for neural networks, whereby the formalisations differ in their focus and variability. Bauer [13] propose graphs for representing the computational processes and functions with *NN* and Fiesler [82] formalises a *NN* by a topology combining a framework of layers and their interconnection scheme. We define a configured *NN* in regard to *NAS* techniques as *FNN* in Definition 5 for a supervised learning (see. Definition 1) task. Further, as we assume a *FNN* structure, we denote the connections between the neurons as a *DAG* g, whereby the nodes of g represent the neurons of the neural architecture.

➢ **Definition 5.** *NN* (*Neural Network*) optimization problem, adopted from Bauer [13] and Fiesler [82]
Let g be a *DAG* (*Directed Arbitrary Graph*) that characterises the connection between a set of nodes $z \in \mathcal{Z}$, which are also referred to as layers. Each node $z \in \mathcal{Z}$ is configured by $\lambda \in \Lambda^{(z)}$ and contains a set of trainable parameters $w \in \mathcal{W}^{(z)}$. Then a *NN* architecture $f : X \rightarrow Y$ can be characterised by $f_{g,\overrightarrow{z},\overrightarrow{\lambda}}$ and the supervised learning goal (see Definition 1) becomes to find the optimal weights $\overrightarrow{w}$ that minimises:

$$\overrightarrow{w}^* \in \underset{\overrightarrow{w} \in \mathcal{W}}{\arg\min}\, \mathcal{V}(\mathcal{L}, f_{g,\overrightarrow{z},\overrightarrow{\lambda}}, \mathcal{D}_{train}, \mathcal{D}_{valid}) \tag{2.12}$$

In addition to the types of networks already discussed for classification, there are other types, such as Auto-Encoders, Extreme Learning Machines, Deep Residual Networks [149] or Graph Neural Networks [206] as well as Spiking Neural Networks [156], whose popularity is increasing. However, for the aim of this work, we rely on *CNN* and *MLP* architectures and refer to Goodfellow et al. [101] for a deeper presentation of different network architectures.

Due to the more complex network structures given by the additional layers compared to single-layer *NN*, it remains unclear how the network can learn, respectively, how the training process works. As stated in Definition 5, the weight should be assigned in such a way that the error function is minimised. The gradient descent method exploits the fact that the negative gradient is defined as the steepest descent direction. This motivates to find a better parameter value by taking small steps in the direction of the negative gradient of the error function $\mathcal{L}$. The algorithm to find suitable weights w^* can be broken down into three essential steps:

- Feed-forward calculation $f_{g,\vec{z},\vec{\lambda}}(\vec{x})$
- Back-propagation from the output to the input layer
- Update of the weights w

In a first step a feed forward calculation is performed on $f_{g,\vec{z},\vec{\lambda}}(\vec{x})$ to estimate the output $\hat{y}$ and thus to calculate the loss $\mathcal{L}(y, \hat{y})$. In the second step, the back-propagation algorithm based on the gradient descent method is applied [202].

The aim is to allocate the weights of the *NN* so that the class membership of the training is modelled in the best possible way. The problem with multi-layered networks is that only the output layer's predicted output values can be checked to see whether the expected class corresponds to the actual class. However, the optimal outputs of the hidden layers are unknown. The back-propagation algorithm, as the name suggests, allows the error of the output layer to be propagated back to the hidden layers [202]. The third step updates the weights of the model $f_{g,\vec{z},\vec{\lambda}}$ based on the gradients received for each layer by the back-propagation step and the loss $\mathcal{L}$. This step is performed by an optimiser. For instance, the *SGD* optimiser adjusts the weights of a model for each $(\vec{x}, y) \in \mathcal{D}$ based on a learning rate and the gradients obtained from the back-propagation step. The learning process is comparatively fast, but the frequent updates with little information gain lead to significant fluctuations in the objective function. Since the performance of *SGD* highly depends on the individual data points, the convergence to the global minimum of the minimisation problem in Definition 5 is made more difficult. In batch gradient descent, the gradients from the back-propagation step are first calculated for all data points within a batch of $\mathcal{D}$ and then the weights are updated based on the resulting mean. Another popular optimization technique to *SGD* is the *Adam* [129]. *Adam* is essentially based on the momentum algorithm and the use of adaptive learning rates to accelerate convergence of the optimization problem and is based on *AdaGrad* proposed by Duchi et al. [70] and *RMSProp* proposed by Hinton et al. [110]. While *AdaGrad* is a modified *SGD* algorithm that adapts the learning rate based on the parameters sparsity, *RMSProp* adapts the learning rate based on a running average of gradients for each weight [202]. Further optimisers are the *SGDHD* optimiser that was introduced by Rumelhart et al. [201] in 1986. It uses as *Adam* a momentum to *SGD* that remembers the weight update at each iteration. The following weight update is applied as a linear combination of gradients from the back-propagation step and the previous updates. This momentum might be beneficial in *online learning*, especially for the adaptation to concept drifts. However, as *Adam* is an extension to *RMSProp*, *AdamW* is an extension to *Adam*, proposed by Loshchilov and Hutter [158] where the L_2 regularisation factor of *Adam* is decoupled from the optimisation process and thus improves the generalization performance.

An enabler for the growing popularity of *NN*s are, on the one hand, the performance and the increasing computational resources available, but also the frameworks that facilitate the realisation of a wide variety of *NN* architectures. Python frameworks such as *Tensor-flow* [1], *Keras* [52] and *PyTorch* [184] in particular offer a wide range of possibilities for the modelling of *DNN*, which can be evaluated against each other using common problems in the machine learning world. While *Tensor-flow* was developed by Google and works on static graph concepts, where the user has to first define the computation graph g of the model and then run the *NN*, *PyTorch* follows a dynamic approach that allows the manipulation of the graph while training. Further, *PyTorch* relies on the *Torch* library [56], that enables the computation of large matrices on *GPU*s, and *Tensor-flow* uses Tensors as core element of the library to perform the matrix multiplications.

In summary, the potential of neural networks becomes apparent due to the multitude of possibilities of their application and the already achieved results, which is why there is also great interest in using *NN*s in the best possible way in practical applications.

2.2.4.6 Ensemble Learning

In many cases, aggregated model's predictions are better than the best individual model. A group of models used to perform an aggregated prediction is called *ensemble* and defined in Definition 6.

➢ **Definition 6.** Ensemble Learning [189]
Given a set of n models $\mathcal{A}^{model} = \{f^{(1)}, \ldots, f^{(n)}\}$ and a data set $\mathcal{D}$, where each model $f : X \to Y \in \mathcal{A}^{model}$ predicts a label $y \in Y$ based on a feature vector $\overrightarrow{x} \in X$. The goal is to learn a joint function

$$\hat{r} : f^{(1)} \times \cdots \times f^{(n)} \times X \to Y \tag{2.13}$$

The idea of *ensemble learning* methods is to select an *ensemble* of models and to combine their predictions. Already in Equation 2.6, we chose from a set of different probabilities for classes the most probable one as final prediction label $\hat{y}$. *Ensemble Learning* techniques thus aim to learn a joint function of predictions made from models that may differ in their predictions. *AutoML* highly relies on *Ensemble Learning* as it often trains an ensemble of *ML* pipelines. The most popular methods are *voting*, *bagging*, *boosting* and *stacking* [94]. In the following, we briefly discuss these techniques.

Voting models aggregate the predictions of each model within an ensemble and predict the value $\hat{y}$ that gets the most votes. Instead of creating separate dedicated models and finding their accuracy, we create a single model that trains by these models and predicts an output based on their combined majority of voting for each output value. This technique is also called *majority voting* and is mostly used when various independent models are at hand. A voting model often achieves higher accuracy than the best model within the ensemble.

Bagging is a technique that enables a set of models by using the same algorithms but training them on different random subsets of $\mathcal{D}_{train}$. A distinction for training the model set is made between the sampling of the data set. When sampling is performed with replacement from the original $\mathcal{D}_{train}$, then this method is called *bagging*. Sampling without replacement is called *pasting* [36].

Boosting is applied when several weak models are combined into a strong learner. Most *boosting* techniques train models sequentially, where each model is trying to correct its predecessor. The most know *Boosting* techniques are *Ada Boost* and *Gradient Boosting* *Ada Boost* corrects its predecessor by paying more attention to the training instances that the predecessor underfitted [94, 87]. By weighting the training instances that the predecessor under fitted and by weighting the models that perform more accurate *Ada Boost* temps to overfit on a training set $\mathcal{D}_{train}$ when using low regularisation models or chaining too many models. *Gradient Boosting* chains models by training the following model on the residual errors made by the predecessor. The prediction $\hat{y}$ is calculated as the sum of the predictions of all models within the chain. *Boosting* techniques have the drawback that they cannot be parallelised while training and do not scale as well as *voting* or *bagging* techniques.

Stacking is the idea of training different models on $\mathcal{D}_{train}$ and aggregating them by training new models based on the predictions made. It was first developed by Wolpert [243], whereby the model that learns from the output of the base models and finally predicts $\hat{y}$ is called *meta learner*. *Stacking* has the advantage that the base models can be trained in parallel, and only the base models and the *meta model* are trained sequentially.

As we have briefly discussed the main ensemble techniques, some broadly used models have emerged again. For instance a *Random Forests* model [230] is an ensemble of decision trees (see Section 2.2.4.1) trained via the *bagging* technique [94]. Furthermore, we want to highlight that these ensemble techniques can be treated as a single model f and thus as an algorithm A parametrised by its hyperparameters $\overrightarrow{\lambda}^{(i)} \in \Lambda_{A^{(i)}}$. Concerning the application of ensemble models in *AutoML*, it can generally be assumed that the ensemble learner contains all hyperparameters of the underlying base models, plus those parameters that the ensemble itself requires. These parameters are, for example, the number of base models for applying *bagging* or the maximum number of the concatenation of models for *boosting* techniques.

Summary

In this section, we presented the concept of a typical *ML* pipeline assuming a supervised learning case, which consists of the steps (i) data preparation (Section 2.2.2), (ii) data preprocessing (Section 2.2.3) and the (iii) *ML* models (Section 2.2.4), following the *KDD* process. The diversity of the algorithms and models, as well as the extensive range of parametrisations, result in numerous development possibilities within the *KDD*, *CRISP-DM* or the *SEMMA* process and thus a large variability to handle a given data set $\mathcal{D}$. Furthermore, we investigated the development of a *ML* pipeline with regard to its automation and thus regarding *AutoML*. But also, in the specific case of *NN*s, we highlighted the broad range of architectures and design decisions that need to be made to successfully apply *NN*s and thus motivate further the application of *NAS*.

2.3 Automation

In this Section, we address the automation of the *ML* pipeline formally defined in Definition 3 and further introduce the concepts of *AutoML* (*Automated Machine Learning*). Further, we introduce different optimisation techniques to automatically search for suitable parametrisations and present existing approaches towards *AutoML*. However, before automating an entire *ML* pipeline, we will focus on the automatic optimisation of hyperparameters $\overrightarrow{\lambda}^{(i)}$ of a single model $A^{(i)}$, that transforms a feature vector $\overrightarrow{x} \in X$ into a target value $y \in Y$. We then introduce in Section 2.3.1 the optimization techniques for *HPO* that can be extended to *AutoML* in Section 2.3.2 and thus utilised for solving the *CASH* problem depicted in Definition 9.

As already stated, many, almost all *ML* systems have hyperparameters that can be set in advance of the training process and thus influence the performance to solve a specific task such as the supervised offline learning task (see Definition 1). The aim of *HPO* is to find the hyperparameters λ^* of a given *ML* model that returns the best performance on a given loss $\mathcal{L}$. We define the *HPO* problem as follows:

➢ **Definition 7.** *HPO* (*Hypterparamter optimization*):
Let $\mathcal{D}$ be a set of data points and $\mathcal{D}_{train}$ and $\mathcal{D}_{valid}$ subsets of $\mathcal{D}$, where $\mathcal{D}_{train} \cap \mathcal{D}_{valid} = \emptyset$. Furthermore, let $f_{\overrightarrow{\lambda}} : X \to Y$ be a model, parametrised by $\overrightarrow{\lambda} \in \Lambda_f$. Λ_f is also referred to as the configuration or search space of f. Then the task is to find the hyperparameters $\overrightarrow{\lambda}^*$ of f that return the best performance on $\mathcal{D}_{valid}$ as measured by a loss $\mathcal{L}$ using a validation protocol $\mathcal{V}(\cdot, \cdot, \cdot, \cdot)$:

$$\overrightarrow{\lambda}^* \in \underset{\lambda \in \Lambda^f}{\arg\min}\, \mathcal{V}(\mathcal{L}, f_{\overrightarrow{\lambda}}, D_{train}, D_{valid}) \tag{2.14}$$

As shown in Equation 2.14 the complexity of *HPO* depends on several variables. The challenges of *HPO* can be resumed as follows [118]:

Evaluation In Definition 7, the possible configurations Λ_f determines the search space of the *HPO* problem and is dependent on the number of options to configure a model f. However, the evaluation of a hyperparameter search space can be very complex since the model is configured in advance of the training process [119, 161] and depends on the complexity of the underlying model.

Complexity The search space Λ_f is often complex and high-dimensional, mainly when comprising a mix of continuous, categorical, but also conditional hyperparameters. Considering i.ex. a model f configured by p parameters with r possible settings each, then the configuration space increases by the complexity of $\mathcal{O}(p^r)$ [16].

Gradient Usually, the models f are seen as *black-boxes*, where the loss function $\mathcal{L}$ has no gradient concerning the hyperparameters of f, and the optimisation process does not consider any information about the inner functional principles of the model. This leads to the inability to apply classical optimisation techniques such as *Gradient Decent*. It becomes even more apparent when assuming data streams and the data needs to be continuously processed, and thus the hyperparameters need to adapt incrementally to the data stream.

Generalisability To obtain models that do not only perform well on a validation data set $\mathcal{D}_{valid}$ another challenge of *HPO* is to configure a model f that generalises well on $\mathcal{D}$. The given data set $\mathcal{D}$ is finite, and thus one cannot optimise for generalisation on $\mathcal{D}_{valid}$.

In many cases, the difficulty with *HPO* lies in the evaluation of the objective function $\mathcal{L}$. Since the hyperparameters are set in advance of a training process and have possibly a significant influence on the model's performance, the search for the hyperparameters λ^* that minimise the loss $\mathcal{L}$ based on validation protocol $\mathcal{V}$ becomes extremely expensive. Exceedingly exact and solvers that search for global optimal configurations in Λ_f are not applicable. Each time we try different hyperparameters, we have to train a model on $\mathcal{D}_{train}$, make predictions on $\mathcal{D}_{valid}$, and then calculate the validation metric to determine the models score. With a large number of hyperparameters and complex models, such as for ensembles (see Section 2.2.4.6), this process quickly becomes intractable to do by hand. To overcome these challenges, we present in the next Section recent heuristics that are commonly used in *HPO*.

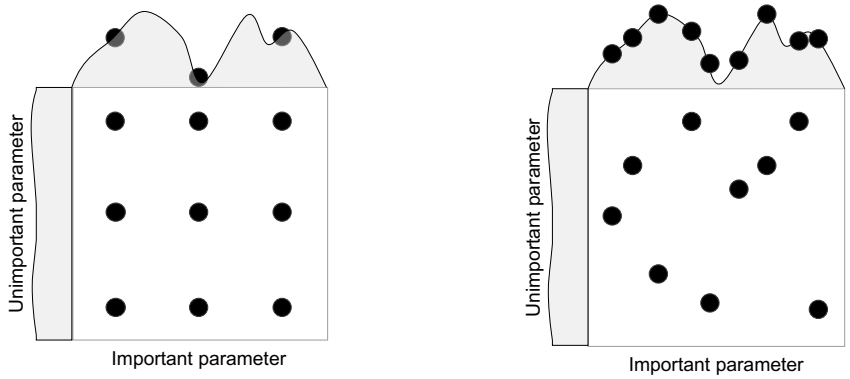
2.3.1 Optimization Techniques

Optimisation heuristics for *HPO* can be characterised by their underlying search technique as well as by their connection to the model f to be optimised. In the latter, when no information about the inner functional principles is accessible, the optimiser is called *black-box optimiser*. These are predominantly limited to *Random Search*, *Grid Search*, *BO*, *GA* (*Genetic Algorithm*) and *Sequential optimization* approaches and are introduced in the following, whereby we introduce in Section 2.3.1.1 the trivial strategies *Random Search*, *Grid Search*, in Section 2.3.1.2 *GA* and in Section 2.3.1.3 *BO* optimisation techniques.

2.3.1.1 Trivial Search Algorithms

The most basic search algorithms for *HPO* are *Grid Search* also referred to as factorial design [171] and *Random Search* [19]. The *Grid Search* exhaustively generates candidates by evaluating the Cartesian product of all parameter configurations specified within Λ_f. This product makes *Grid Search* suffer from the curse of dimensionality since the number of evaluations increases exponentially with the growth of the configuration space Λ_f [16]. *Grid Search* is simple to implement, and when not bound to a limited number of evaluations, it generates a globally optimal solution within Λ_f. Furthermore, it can easily be parallelised and is reliable in low dimensional configuration spaces. This has even led to the common use of *Grid Search* in *HPO* for *ML* systems [94] as well as in *NAS* [19, 143, 110].

In *black-box optimization Random Search* is a useful baseline for *HPO*, since it makes no assumption on the *ML* model being optimised [118]. It samples configurations $\overrightarrow{\lambda}$ at random until a particular budget for the search is exhausted. Especially when some hyperparameters are more critical than others, *Random Search* finds better models than *Grid Search*. Figure 2.7 illustrates how parameter grids and uniformly random parameters differ, when a parameter p_1 has high and p_2 a low influence on a function $f(p_1, p_2)$. A grid of parameters leads to an inefficient coverage of the exploration space. In contrast, the choice of random parameters are more unevenly distributed and thus provide a more evenly distributed coverage. When considering a budget of B for evaluations in both optimisation techniques, both algorithms have no



(a) Grid Search

(b) Random Search

Figure 2.7: Comparison *Grid Search* and *Random Search* with a budget of nine trials for optimizing a function $f(p_1, p_2) = g(p_1) + h(p_2) \simeq g(p_1)$. Above each square $g(p_1)$, and on the left of each square $h(p_2)$ with low effect on $f(p_1, p_2)$ is plotted. This Figure is based on Bergstra and Bengio [19].

guarantee to generate a globally optimal solution within the configuration space. By considering a budget Bergstra and Bengio [19] claim, that *Random Search* finds better models by effectively searching a larger configuration space. A random sampling of hyperparameters can be used to initialise *GA* and, in particular,

EA that further explore the configuration space Λ_f. These search heuristics are depicted in more detail within the following Section.

2.3.1.2 Evolutionary Algorithms

Population-based methods such as *EA* (*Evolutionary Algorithm*) create a set of configurations, and improve this population by applying local perturbations also referred to as mutations. *EA* can be split into (i) *GA* (*Genetic Algorithm*), (ii) *GP* (*Genetic Programming*), (iii) *ES* (*Evolutionary Strategy*), (iv) *EP* (*Evolutionary Programming*) and (v) *PSO* (*Particle Swarm Optimization*) [71, 214, 241]. They differ primarily in their operational scheme depicted in Figure 2.8 and their encoding of the parameter space Λ for the resulting search space of the algorithm. This encoding is particularly useful if the representation of a possible solution $\overrightarrow{\lambda}$ can be significantly simplified. However, the historically emerged search techniques are often used as synonyms

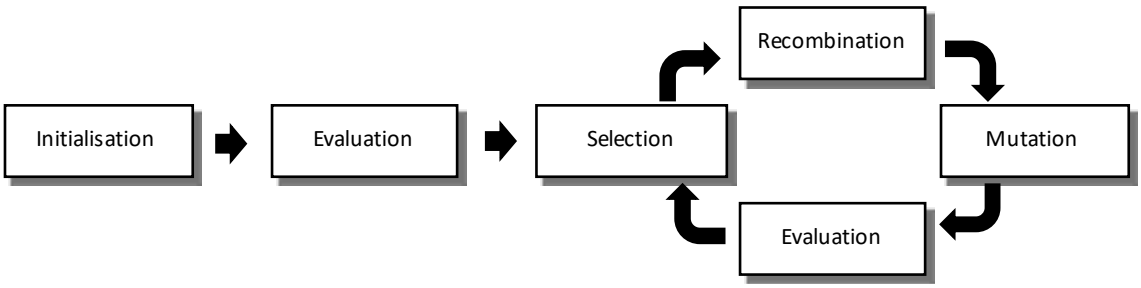


Figure 2.8: The procedure of *EA* usually consists of an (i) initialisation, (ii) evaluation phase and a generation loop containing a (iii) Selection, (iv) Recombination, (v) Recombination and an (vi) Evaluation step that is run through until a termination criterion is fulfilled.

for the entire field of evolutionary algorithms. An abundance of other methods and an unmanageable number of combinations have led to terms conflating. Whitley [241] provide a theoretical comparison between the classical bit-coded *GA*, *ES* and *EP*. As our approach towards a *online AutoML* framework relies on *EA*, we depict and characterise the broad operational scheme in Figure 2.8 in deeper detail in the following:

Initialisation The initialisation of an *EA* contains the first generation of solution candidates and is mainly generated randomly. These candidates are also referred as *population* or *swarm* in *PSO* and are subsequently evaluated. Considering the Definition 7 *GA* require a genotype-phenotype mapping to enable the search within categorical as well as continuous hyperparameters [113]. *ES* and *PSO* [31] can handle the configuration space Λ directly so that the configuration space and the search space of the algorithm are identically. *GP* initialise and represent an individual (configuration $\overrightarrow{\lambda}$) of the population within a tree structure and *EP* within finite-state machines. Notably, in *PSO*, the initialisation of a swarm has a significant impact on the optimisation performance since all *particles* (individuals configured by $\overrightarrow{\lambda}$) of the swarm are updated within the search process.

Selection This step is used to select solution individuals for recombination (parental selection) and determine the next generation of the underlying algorithm. To select the most vital individuals, for each individual, a *fitness* is calculated, which forms the reproduction success of the individuals. In *HPO* this fitness is represented by the loss $\mathcal{L}$. Common methods for selection are (i) *roulette wheel*, (ii) *rank*, (iii) *steady state*, (iv) *tournament*, and (v) *elitism* selection [160]. In *roulette wheel* selection, the probability of choosing an individual for breeding of the next generation is proportional to its fitness. *Rank* selection selects individuals based on their fitness rank within the population and thus considers an equal share of being set. In *steady state* selection, only a few good individuals from the population are selected for creating a new offspring. Poorly performing individuals are removed for the next generation. *Tournament* selection incorporates several comparisons based on their fitness among randomly chosen individuals from the population. The winner of each competition is used for the recombination step. In *elitism* selection, well-performing individuals in the population are used for the next generation without any changes.

Recombination The recombination or *crossover* step [160] produces an offspring population by randomly combining randomly the selected individuals from the population. This step aims to transfer traits (hyperparameters) from the selected individuals that have positive effects on the configuration and thus the fitness $\mathcal{L}$. Due to the diverse structure (binary, categorical or continuous) of parameters within configuration space, Λ different algorithms are carried out to perform the recombination step. For a genetic recombination $k - point$ crossover [160] is a commonly applied for recombination, where k crossover points are chosen from a bit array representation of $\overrightarrow{\lambda}$. The bits between the crossover points are swapped between the selected individuals to generate a new configuration $\overrightarrow{\lambda}$.

Mutation After having generated a new configuration $\overrightarrow{\lambda}$, a mutation step is applied, where the design of the model is randomly changed. This step is usually controlled by a mutation rate that gives the probability that an individual of the newly generated population is mutated. In a binary representation of $\overrightarrow{\lambda}$, a mutation can be applied by randomly flipping bits of the array. For continuous variables the mutation can be applied using a $\mathcal{N}(0, \sigma)$ distribution.

Evaluation After the initialisation or the mutation step (see Figure 2.8), the algorithm contains a population of models (individuals) where the fitness or the performance is unknown. In this step, these individuals are evaluated, and their fitness is calculated. For *HPO* this step calculates the fitness based on a validation protocol $\mathcal{V}(\cdot, \cdot, \cdot, \cdot)$ by training the model $f_{\overrightarrow{\lambda}}$ on a training data set $\mathcal{D}_{train}$ and validating it based on $\mathcal{D}_{valid}$. The computation of the fitness (see Definition 7) can be summarised by performing $\mathcal{V}(\mathcal{L}, f_{\overrightarrow{\lambda}}, \mathcal{D}_{train}, \mathcal{D}_{valid})$.

A broadly used *EA* is *NSGA-II* that is capable of finding multi-objective solutions by performing the basic steps of an *EA* (see Figure 2.8) and generating a new population by selecting from a combined population of the old population. Further, it performs a tournament selection, recombines the selected individuals and mutates them into a newly generated population. *NSGA-II* has successfully been applied in *AutoML* by Olson et al. [177, 144] in *TPOT*. A prominent example for *ES* is the *CMA-ES* [106], where the idea is to learn the shape of the search space during evolution. This approach aims to increase the probability of previously successful steps. The covariance matrix of the distribution is changed in such a way that the probability of the selected step of the last generation is increased [214]. It is one of the most competitive [118] *black-box* optimization algorithms. In *NAS*, Real et al. [198] apply a *regularized evolutionary* approach, that skips the recombination step by only copying and mutating the best performing neural architectures. The oldest

individual is removed from the population of neural architectures. Similar to *Grid Search* and *Random Search* the search for the best hyperparameter configuration $\overrightarrow{\lambda}^*$ of a model f_λ can be parallelised in *EA*'s by calculating $\mathcal{V}(\mathcal{L}, f_{\overrightarrow{\lambda}}, \mathcal{D}_{train}, \mathcal{D}_{valid})$ for each individual of the configuration in a parallel manner.

In this section, we presented the foundations for *EA* techniques for solving the *HPO* problem in Definition 7. In the following we present *BO* as another *HPO* technique that can also be applied in *AutoML* and *NAS*.

2.3.1.3 Bayesian Optimization

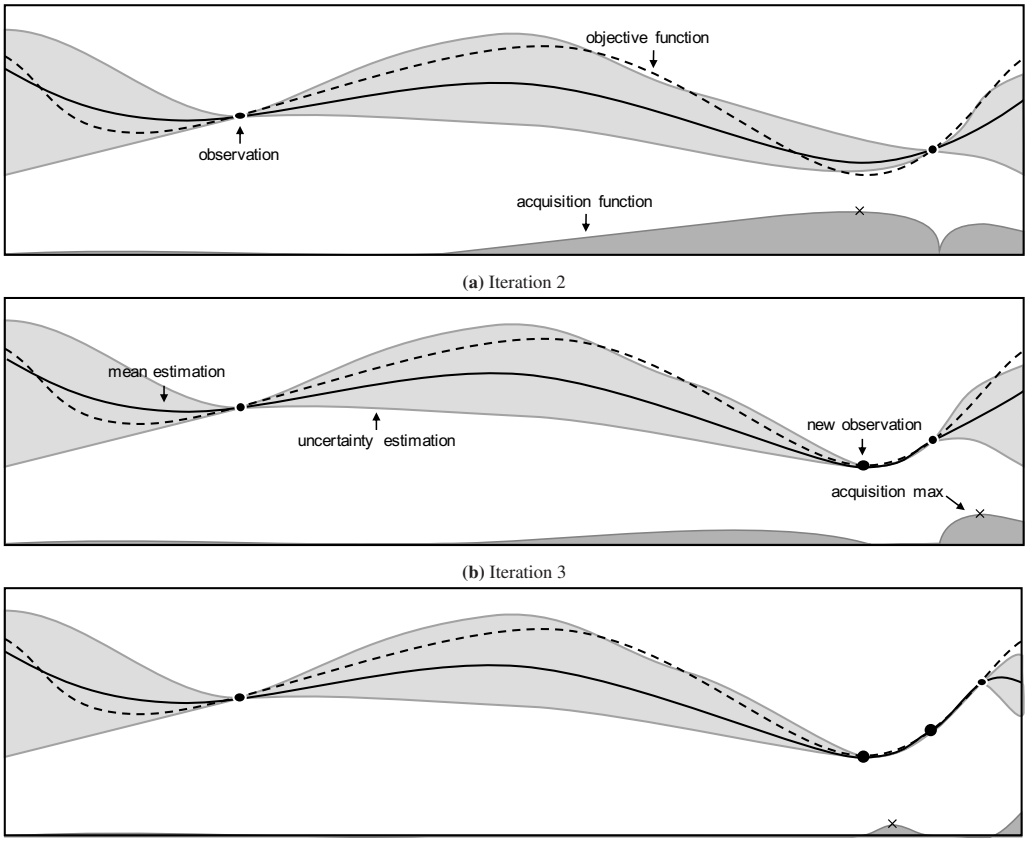
Challenges in *HPO* are the evaluation time and complexity of the underlying configuration space Λ. *BO* (*Baysian Optimization*) as well as *EA* approaches do not assume any knowledge of the underlying model and are thus suited for *black-box* optimisation. It iteratively generates a surrogate model [118, 117] that quantifies the uncertainty and thus estimates the utility of a configuration $\overrightarrow{\lambda}$. This estimation is used to decide for the next configurations $\overrightarrow{\lambda}$ to be evaluated by the validation protocol $\mathcal{V}(\mathcal{L}, f_{\overrightarrow{\lambda}}, \mathcal{D}_{train}, \mathcal{D}_{valid})$. *BO*, therefore consists of two components: (i) the surrogate model for iteratively modelling the objective function of *HPO* and (ii) an acquisition function that decides where to sample next [117]. Compared to the execution of the validation protocol, the acquisition function is cheap to compute and can, therefore, effortlessly be optimised. Thus, it trades between exploitation and exploration. The acquisition function is high where the surrogate model predicts a high objective score (exploitation) and where the uncertainty of the surrogate model is high (exploration). In *BO* a *Gaussian process* is commonly applied to build a surrogate model. The *Gaussian process* $\mathcal{GP}(m(\overrightarrow{\lambda}), k(\overrightarrow{\lambda}, \overrightarrow{\lambda}'))$ is specified by the mean function of all already known configuration performances $m(\overrightarrow{\lambda})$ and a covariance function $k(\overrightarrow{\lambda}, \overrightarrow{\lambda}')$ [37, 118]. The mean predictions $\mu(\overrightarrow{\lambda})$ can be expressed by the vector of covariances $\mathbf{k}_*$, the covariance matrix $\mathbf{K}^{-1}$ of all previous observations and the achieved scores y_s [118] for the previous observations from the validation protocol. The mean and variance predictions can be obtained by:

$$\mu(\overrightarrow{\lambda}) = \mathbf{k}_*{}^T \mathbf{K}^{-1} y_s \qquad\qquad \sigma^2(\overrightarrow{\lambda}) = k(\overrightarrow{\lambda}, \overrightarrow{\lambda}') - \mathbf{k}_*{}^T \mathbf{K}^{-1} \mathbf{k}_*$$

The choice of the covariance function $k(\overrightarrow{\lambda}, \overrightarrow{\lambda}')$ considerably influences σ^2 and thus indicates the essential characteristics of the process. However, common choices for this function are the Mátern kernel [118] or a squared exponential kernel [37] function. To determine a well-balanced trade-off between exploitation and exploration of the acquisition function concerning the *Gaussian process*, the *expected improvement* [126] is a common choice, where all possible improvements by the surrogate model (*Gaussian process*) are weighted.

In Figure 2.9, we depict adapted from [37, 118] the iterations of the *BO* process, whereby the *Gaussian prior* encapsulates all assumptions or information that where already observed by evaluating different hyperparameter configurations $\overrightarrow{\lambda}$. If new hyperparameter configurations are made, the prior belief is updated based on the new evaluations (also referred to as observations) made. The result is a *Gaussian posterior*. The optimisation procedure iteratively explores the search space Λ by optimising the acquisition function and thus finding a functional trade-off configuration $\overrightarrow{\lambda}$ between exploitation and exploration. This sample configuration (in Figure 2.9 referred to as observation) is evaluated on the objective function (see Definition 7).

The *Gaussian prior* is updated based on the sampled configuration and its score from the objective function $\mathcal{L}$. This procedure is repeated until a stopping criterion, such as a maximum number of observations, is fulfilled.



(a) Iteration 2

(b) Iteration 3

(c) Iteration 4

Figure 2.9: Iterations of the *BO* process adapted from [37, 118] based on one continuous hyperparameter $\overrightarrow{\lambda}$ (x-axis), where the goal is to solve the *HPO* problem (see Definition 7) and thus to minimise the loss $\mathcal{L}$ of a model $f_{\overrightarrow{\lambda}}$ parametrised by $\overrightarrow{\lambda}$ using a validation protocol $\mathcal{V}(\cdot,\cdot,\cdot,\cdot)$. The objective function is depicted as a dashed black line. The optimisation process uses a *Gaussian process* to build a surrogate model, where the predicted mean is shown as a black line, and the uncertainty estimation is represented by the grey tube. The acquisition function (bottom) is high where the surrogate model estimates a low mean (exploitation) and where the uncertainty of the surrogate model is high (exploration).

Approximating the the objective function by a surrogate model has led to new state-of-the are results in *HPO* and thus the application on *AutoML* [81, 80] and *NAS* [218, 217, 166]. *BO* is exceptionally well suited for *HPO* problems where the configuration space is small and continuous. However, it scales cubically [118] with the number of observed data points (evaluations), and it becomes increasingly complex when the configuration space Λ is highly dimensional. The iterative process has a further disadvantage in that it is difficult to parallelise. To overcome these drawbacks Hutter et al. propose *SMAC*, a framework for *BO* that incorporates random forests.

In this section, we presented the main optimisation methods used in the field of *HPO*. We presented trivial strategies, *EA* and *BO* based *HPO* techniques, that already have successfully been applied in *HPO*. Furthermore, we highlighted each of the approaches, their advantages and disadvantages. In *HPO*, we have so far addressed the problem of finding a suitable model configuration $\overrightarrow{\lambda}$. However, to automate the configuration of a *ML* pipeline (see Figure 2.4) the search space remains more complex, as parameters of available models that are not used within the configured *ML* pipeline do not need to be set and thus an extension to *HPO* is necessary to incorporate a *ML* pipeline, but also to search within *NAS* search spaces. The extension of *HPO* to configure *ML* pipelines is referred to as *AutoML* and to configure neural architectures it is referred to as *NAS*. Both are depicted in the following sections.

2.3.2 Automated Machine Learning

So far, we have addressed the problem of finding a suitable model configuration $f_{\overrightarrow{\lambda}^*}$ by applying *HPO*. *AutoML* (*Automated Machine Learning*), however, aims to automate a *ML* pipeline $\mathcal{P}$ [81] comply containing the steps (i) data preparation, (ii) feature engineering and (iii) modelling algorithm (see Figure 2.4). In this section, we formalise in Definition 8 the pipeline creation problem that enable the automation of the *ML* pipeline as well as the underlying *CASH* Problem that is commonly used in recent *AutoML* frameworks. Subsequently, we present common implementations for *AutoML*.

In Definition 3 we defined a *ML* pipeline as a triplet $(g, \overrightarrow{A}, \overrightarrow{\lambda})$, where $g \in G$ represents within a *DAG* the pipeline's structure, $\overrightarrow{A} \in \mathcal{A}$ the employed algorithms and $\overrightarrow{\lambda} \in \Lambda$ their configurations. Automating this configuration of a *ML* pipeline, however, incorporates finding a suitable structure $g^* \in G$ and choosing the right algorithms $\overrightarrow{A}^* \in \mathcal{A}$. Following the definition from Thornton et al. [228] a *ML* pipeline structure $g \in G$ can be modelled as an arbitrary *DAG*, where each node represents an algorithm $A_{cleaning}, A_{feature}$ and A_{model} from each step. The problem to find a suitable pipeline $\mathcal{P}$ that incorporates the search for a suited structure $g^* \in G$ can be defined as follows:

➢ **Definition 8.** Pipeline Creation Problem, adopted from [253, 228]:
Let $\mathcal{A} = \{A^{(1)}, \ldots, A^{(R)}\}$ be a set of step independent algorithms, and let the hyperparameters of each algorithm $A^{(j)}$ have a domain $\Lambda^{(j)}$. Further, let $\mathcal{D}_{train}$ be a training and $\mathcal{D}_{valid}$ be a validation set with $\mathcal{D}_{valid}$. Let $\mathcal{L}(\mathcal{P}_{g,\overrightarrow{A},\overrightarrow{\lambda}}(\mathcal{D}_{train}), \mathcal{D}_{valid})$ denote the metric that algorithm combination $\mathcal{P}$ achieves on $\mathcal{D}_{valid}$ when trained on $\mathcal{D}_{train}$ with hyperparameters $\overrightarrow{\lambda}$. Then the pipeline creation problem is to find a pipeline structure g^*, the joint algorithm combination $\overrightarrow{A}^*$ and the hyperparameter setting $\overrightarrow{\lambda}^*$ that minimises the metric $\mathcal{L}$ by applying a validation protocol $\mathcal{V}(\cdot,\cdot,\cdot,\cdot)$:

$$g^*, \overrightarrow{A}^*, \overrightarrow{\lambda}^* \in \underset{g \in G, \overrightarrow{A} \in \mathcal{A}^{|g|}, \overrightarrow{\lambda} \in \Lambda}{\arg\min} \mathcal{V}(\mathcal{L}, \mathcal{P}_{g,\overrightarrow{A},\overrightarrow{\lambda}}, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{valid}}) \tag{2.15}$$

The problem of finding a suitable combination of the presented algorithm steps $A_{cleaning}, A_{feature}$ and A_{model} without considering a variable pipeline structure $g \in G$ is often reduced to an algorithm selection and parametrisation problem. Whereby this reduced approach is in many cases [228, 81, 118] also referred to as *CASH* problem and further defined in Definition 9. Instead of solving the pipeline creation problem defined in Definition 8 that considers the optimisation of the pipeline structure, the majority of *AutoML* frameworks solve the *CASH* Problem. Neglecting the pipeline structure within the *CASH* Problem dramatically reduces the complexity of the search problem [253]. However, this relief may also lead to inferior performances for complex data sets that require multiple, from the fixed structure differing, algorithms steps.

➢ **Definition 9.** CASH problem, adopted from [228].
Let $\mathcal{A} = \{A^{(1)}, \ldots, A^{(R)}\}$ again be a set of step independent algorithms, and let the hyperparameters of each algorithm $A^{(j)}$ have a domain $\Lambda^{(j)}$. Further, let $\mathcal{D}_{\text{train}}$ be a training and $\mathcal{D}_{\text{valid}}$ be a validation set, which is split into K cross-validation folds $\{\mathcal{D}^{(1)}_{\text{train}}, \ldots, \mathcal{D}^{(K)}_{\text{train}}\}$ and $\{\mathcal{D}^{(1)}_{\text{valid}}, \ldots, \mathcal{D}^{(K)}_{\text{valid}}\}$. Let $\mathcal{L}(\mathcal{P}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}(\mathcal{D}^{(i)}_{\text{train}}), \mathcal{D}^{(i)}_{\text{valid}})$ denote the metric that algorithm combination $\mathcal{P}^{(j)}$ achieves on $\mathcal{D}^{(i)}_{\text{valid}}$ when trained on $\mathcal{D}^{(i)}_{\text{train}}$ with hyperparameters $\overrightarrow{\lambda}$. Then the CASH problem is to find the joint algorithm combination and hyperparameter setting that minimises the metric:

$$\overrightarrow{A}^*, \overrightarrow{\lambda}^* \in \underset{\overrightarrow{A} \in \mathcal{A}^{|g|}, \overrightarrow{\lambda} \in \Lambda}{\arg\min} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(\mathcal{P}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}(\mathcal{D}^{(i)}_{\text{train}}), \mathcal{D}^{(i)}_{\text{valid}}) \tag{2.16}$$

When considering a fixed pipeline structure [228, 134, 223, 81] the pipeline creation problem is reduced to selecting suitable algorithms $\overrightarrow{A}^* \in \mathcal{A}^{|g|}$ with their configuration $\overrightarrow{\lambda} \in \Lambda$ by minimising the loss $\mathcal{L}$ within a validation protocol $\mathcal{V}(\cdot, \cdot, \cdot, \cdot)$. In addition to the *HPO* problem (Definition 7) and the pipeline creation problem defined in Definition 8, the *CASH* Problem already incorporates a validation protocol $\mathcal{V}$ that split the data set $\mathcal{D}$ into K folds to evaluate a pipeline $\mathcal{P}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}$ on the given loss $\mathcal{L}$ on the averaged sum of losses. However, many *AutoML* frameworks rely on the *CASH* Problem and thus are limited to the fixed and predefined pipeline structure, such as illustrated in Figure 2.4 [253].

The most established frameworks for *AutoML* are *Auto-Weka* [228], *Auto-Weka 2.0* [135], *autosklearn* [81], *autosklearn 2.0* [80], *TPOT* [177] and *h2o* [147]. Zöller and Huber [253] compare these frameworks by their (i) underlying solver, their (ii) fixed or variable structure and their (iii) configuration possibilities. Furthermore, recent developments in *DL* lead to the integration of *NAS* into some *AutoML* solvers. In Table 2.2, we summarise the differences for most popular *AutoML* frameworks.

Table 2.2: Comparison of different *AutoML* frameworks adopted from Zöller and Huber [253]

Framework	Solver	Structure	Ensemble	NAS	Parallel	Time
Auto-Weka [228, 135]	SMBO [117]	Fixed	✓	×	×	✓
Auto-Sklearn [81, 80]	SMBO [117]	Fixed	✓	×	✓	✓
TPOT [177]	*GP* [85]	Variable	✓	✓	✓	✓
Hypersklearn [134]	Hyperopt [20]	Fixed	×	×	×	✓
ATM [223]	Bandit	Fixed	×	×	✓	×
h2o [147]	Grid Search	Fixed	✓	✓	✓	✓

In Table 2.2, we compare different *AutoML* frameworks by their solver, whether they consider a fixed or variable structure g, employ ensemble learning (see Section 2.2.4.6), implement *NAS* techniques, or whether they are capable of executing the search for suitable pipelines in a parallel manner or by consideration of a time budget. These characteristics provide the main differences for the most established *AutoML* frameworks [253] considered by the number of citations and stars on GitHub[1]. To depict further differences, we present

[1] www.github.com, accessed January 30, 2023

in the following a short overview for the the *AutoML* solver presented in Table 2.2 adapted from Zöller and Huber [253].

Auto-Weka Thornton et al. [228] introduced in their work not only the *CASH* problem, but the framework *Auto-Weka* that aims to solve the *CASH* problem. *Auto-Weka* showed on a broad range of classification tasks that combined algorithms selection, and hyperparameter optimisation techniques often perform much better than standard selection and hyperparameter optimisation methods. It is based on the *Weka* [104] framework, and in its first implementation, it considered a subset of 37 applicable classifiers that also incorporates the selection of ensemble learners. *Auto-Weka* as well as *Weka* [104] are implemented in *Java* and consider a fixed pipeline structure. Furthermore, *Auto-Weka* [228] and *Auto-Weka 2.0* [135] use the *SMAC* [117] solver to optimise the search space.

Auto-Sklearn provides in its first version [81] very similar functionalities to *Auto-Weka* but is implemented in *Python* and based on *Scikit-learn* [185]. *ML* pipelines are tuned in a semi-structured design using the *SMAC* algorithm proposed by Hutter et al. [117].

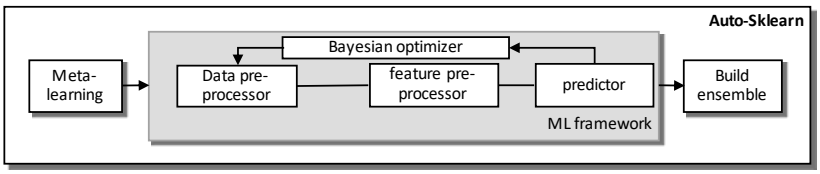


Figure 2.10: *Auto-Sklearn* framework [81]

First, the algorithm selects from a fixed set of cleaning steps containing (i) categorical encoding, (ii) imputation and (iii) scaling an appropriate preprocessing step. An optional feature preprocessing step and a suited predictor (classifier or regressor) is configured and executed via the *SMAC* optimiser. The prediction step incorporates the building of ensembles. The selection of a suited predictor incorporates the configuration of ensembles, and since the (i) data preprocessing and (ii) feature preprocessing steps are optional steps, *Auto-Sklearn* solves the pipeline creation problem depicted in Definition 8 in a semi-structured manner. *Auto-Sklearn* 2.0 [80] enables *AutoML* to work well on large data sets under rigid time limits. Figure 2.10 depicts the *AutoML* framework of *Auto-Sklearn* applying *BO* (*SMAC*) as solver for the underlying *CASH* problem.

TPOT is as well as *Auto-Sklearn* based on *sklearn* [177] but incorporates *GP* algorithms to optimise a *ML* pipeline with a variable, tree-based structure. The application of *GP* as underlying *HPO* technique has the drawback, that *TPOT* can only handle categorical parameters for configuring a *ML* pipeline. However, to combine the (i) preprocessing, (ii) decomposition, (iii) feature selection and (iv) modelling operators from the *sklearn* environment into a flexible pipeline structure, *TPOT* processes the initial data set $\mathcal{D}$ in a parallel manner by copying the data set, processing it and merging the data set within a combination operator. This enables *TPOT* to handle a variable pipeline structure. Recent developments in this framework led to the integration of (simple) *NAS* techniques depicted in more detail in Section 2.3.3. Furthermore, the *GP* optimiser enables to control the complexity and time budget as well as the greediness by configuring a population size, a number of generations and a time budget.

Hypersklearn was introduced by Komer et al. [134] and was build around the *Hyperopt* and *sklearn* frameworks [20] and implemented in *Python*. In comparison to *Auto-Sklearn* and *TPOT Hypersklearn* considers a fixed pipeline structure containing exactly one pre-processing and one classification or regression step. Since *Scikit-learn* provides a modular structure, *Hypersklearn* is a thin wrapper around the *Hyperopt* framework that follows the *Scikit-learn* design principles, as well as a configuration space. The pipeline structure is fixed to one preprocessor and one classification or regression algorithm. Parallelisation of the evaluation is not available.

ATM Swearingen et al. [223] propose a bandit based approach with a fixed pipeline structure based on *sklearn*. It enables users to simply upload a data set, choose a subset of modelling methods and execute the underlying *AutoML* framework. This collaborative approach enables storing meta-features and thus predetermining suitable *ML* pipelines before starting the optimisation procedure. Furthermore. *ATM* is a distributed approach for *AutoML* that allows to simultaneously generating multiple *ML* pipeline candidates.

h_2o LeDell and Poirier [147] propose a highly scalable *AutoML* framework that is available in *R*, *Python*, *Java* and *Scala* and can thus be used seamlessly within a diverse team of data scientists. Furthermore, h_2o is promoted as *ML* platform where the underlying grid search approach evaluates a fixed *ML* pipeline structure and ranks the pipelines being assessed within a leader-board, which can be easily exported for use in a production environment. It further applies *Grid Search* to build the leader-board.

The differences of frameworks can be summarised by their underlying solver, as well as based on their ability to optimise *ML* pipelines with a fixed or variable structure (see Table 2.2). While *Auto-Weka 2.0* and *autosklearn* use a *Random Forest* and Gaussian processes [117, *SMAC*], *TPOT* employs an *EA* and h_2o a *Grid Search* approach, resulting in different features such as ensembles of different pipelines, or a parallel evaluation. However, their differences also result from the different goals pursued by the frameworks. While *Auto-Sklearn*, *TPOT*, *Hypersklearn* have the goal to automate the *ML* pipeline considering a fixed or variable structure, *ATM* considers a collaborative approach in order to enable *meta-learning* techniques and h_2o's focus relies on the scalability of an entire *ML* platform that supports end-users by a user interface.

In this section, we formalised the *ML* pipeline creation problem in Definition 8 and the *HPO* problem in Definition 7. Further, we located the *CASH* problem (Definition 8) as a reduced pipeline creation problem and highlighted the differences between the problems. With regard to the application of *AutoML* libraries within our approach towards a utility-based adaptation, we presented existing *AutoML* frameworks. In Table 2.2, we further summarise the differences and similarities of selected *AutoML* systems. In the following, we present the foundations *NAS* (*Neural Architecture Search*) that enable the optimisation towards suitable neural architectures.

2.3.3 Neural Architecture Search

The success of *AutoML* in a broad range of supervised learning tasks, as well as the success of *DL*, led to the development of *NAS* (*Neural Architecture Search*) as a subfield of *AutoML*. *NAS* has gained great attention in the *ML* community after Zoph and Le[255] obtained competitive performance on the *CIFAR*-10 and further image classification benchmark data sets. Zoph and Le [255] applied in contrast to the *HPO* techniques presented in Section 2.3.1 a *RL* based optimisation technique, where an agent learns to configure a suitable neural architecture by obtaining an training based on the metric loss of evaluated neural architectures from

an so called environment. While Zoph and Le [255] used vast computational resources to achieve the results (800 *GPU*s), and have pushed the research into more efficient search strategies. A wide variety of methods have been published in quick succession to reduce computational costs and achieve further performance improvements on the benchmark data sets. This success has further been accompanied by rising demand for architecture engineering, where increasingly more complex neural architectures are designed manually [74]. Similar to *AutoML*, *NAS* methods can be categorised according to their (i) search space, (ii) strategy and (iii) performance estimation strategy. As depicted in Figure 2.11, the search strategy selects a model from a predefined search space $(G, \mathcal{Z}, \Lambda)$. This model is passed to a performance estimation strategy that returns the performance (loss) to the search strategy to give a search direction and enable further optimisation. However, already here, the similarity to *AutoML* techniques becomes apparent.

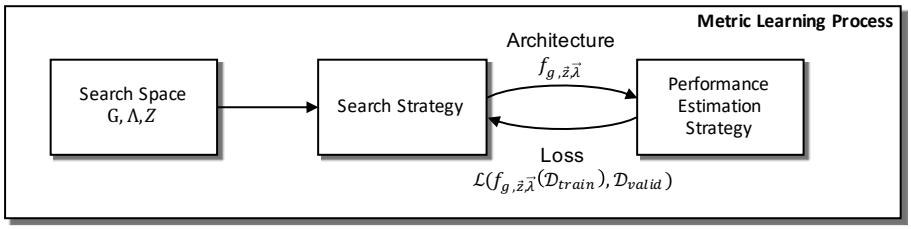


Figure 2.11: Illustration of the *NAS* methodology adapted from Elsken et al. [74] containing the interaction of a search space $G, \mathcal{Z}, \Lambda$, a search strategy, and a performance estimation strategy.

To automate the search for suitable structures, the concept of *NAS* has been proposed by Pham et al. [188] and has shown great success on various data sets. Current research in *NAS* e.g. [125, 5, 122, 11] have shown that *NAS* frameworks are able to outperform hand crafted state-of-the-art architectures and to set new benchmarks on established data sets. We formalise in Definition 10 accordingly to Figure 2.11 and Definition 5 the *NAS* problem. As stated in Section 2.2.4.5, Bauer [13] formalise a *NN* as a *DAG* that transforms an input X to an output Y with a set of nodes $z \in \mathcal{Z}$. Thus, based on the *CASH* problem in Definition 9 and Definition 5 for *NN*s, we formalise in Definition 10 *NAS* as follows

➢ **Definition 10.** *NAS* Problem, adopted from Jin et al. [125]:
Let $\mathcal{D}_{train}$ be a train and $\mathcal{D}_{valid}$ be a validation data set, which is split into K cross validation folds $\{\mathcal{D}^{(1)}_{train}, ..., \mathcal{D}^{(K)}_{train}\}$ and $\{\mathcal{D}^{(1)}_{valid}, ..., \mathcal{D}^{(K)}_{valid}\}$. Furthermore, let $f_{g,\overrightarrow{z},\overrightarrow{\lambda}} \in \mathcal{F}$ be a neural network, configured by a graph structure $g \in G$ and a set of nodes $z \in \mathcal{Z}$, with a parametrisation $\overrightarrow{\lambda} \in \Lambda^z$ Let $\mathcal{L}(f_{g,\overrightarrow{z},\overrightarrow{\lambda}}(D^{(i)}_{\text{train}}), D^{(i)}_{\text{valid}})$ denote the loss function a neural network $f_{g,\overrightarrow{z},\overrightarrow{\lambda}}$ achieves on $D^{(i)}_{valid}$ when trained on $D^{(i)}_{train}$. Then the neural architecture search problem is to find the neural network $f_{g^*,\overrightarrow{z}^*,\overrightarrow{\lambda}^*}$ that minimises the loss:

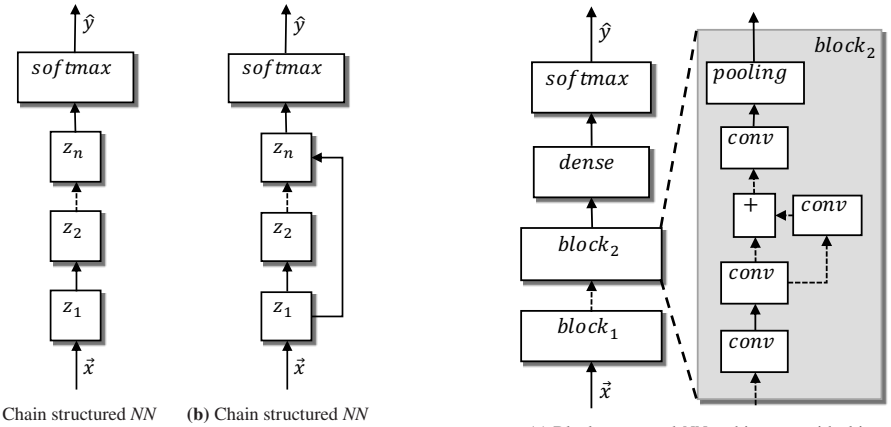
$$g^*, \overrightarrow{z}^*, \overrightarrow{\lambda}^* \in \underset{g \in G, \overrightarrow{z} \in \mathcal{Z}, \overrightarrow{\lambda} \in \Lambda^{|g|}}{\arg\min} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(f_{g,\overrightarrow{z},\overrightarrow{\lambda}}(\mathcal{D}^{(i)}_{train}), \mathcal{D}^{(i)}_{valid}) \tag{2.17}$$

As already referred to within the *CASH* problem, Definition 10 splits a given data set $\mathcal{D}$ into K - folds to average the sum of losses and to use all data available within $\mathcal{D}$ to train and validate a network $f_{g,\overrightarrow{z},\overrightarrow{\lambda}}$. By choosing this validation protocol the *performance estimation strategy* in Figure 2.11 is influenced by the number of epochs to update the weights of a given model $f_{g,\overrightarrow{z},\overrightarrow{\lambda}}$. Furthermore, *NAS* approaches differ

in their search space they search in and the underlying search strategy. The differences are depicted in the following.

2.3.3.1 Search Space

Comparable to *AutoML*, there can many different *search strategies* be applied to explore a *search space* $G, \mathcal{Z}, \Lambda$. However, the choice of the underlying search space has a high impact on the performance and, thus, on the comparability of the underlying search strategy. In *NAS*, however, the search space differs in the connection types of different layers, depicted in Figure 2.12.



(a) Chain structured *NN* architecture

(b) Chain structured *NN* with skips

(c) Block structured *NN* architecture with skips

Figure 2.12: Illustration of different architectures when a) concatenating different layers, b) concatenating different layers with skip connections, and c) concatenating configurable blocks of layer structures adopted from Elsken et al. [74]. Each architecture has configurable (black arrows) and fixed (dashed arrows) connections to predict from an input $\vec{x}$ an output $\hat{y}$ by processing the input within operators associated with nodes $z \in \mathcal{Z}$.

When concatenating i.ex. various layers such as fully connected (dense) or *CNN* layers to a *FNN* the search space is parametrised by the (i) maximum number of layers, (ii) the layer type, or the operation of the layer executes, e.g. normalisation, pooling, convolution and (iii) the hyperparameters (λ) associated with this operation [74]. In Figure 2.12a, we illustrate the approach that concatenates different layers without skip connections. These approaches optimise mainly *CNN* architectures [11, 220, 46] by concatenating *CNN* layers an configuring e.g. the number of filters, kernel size and strides or the number of units for fully-connected *NN* (*MLP*) networks [167].

However, modern *NN* architectures are not represented by the simple concatenation of layers and include additional types of architectures, e.g. skip connections depicted in Figure 2.12b or multi-branch architectures [74]. These more complex architectures have higher degrees of freedom and thus cover a more extensive search space $G, \mathcal{Z}, \Lambda$. They enable to map complex residual structures, where previous layer outputs are summed [107] or concatenated [115]. Brock et al. [38] propose a configuration space that consists of a large set of memory blanks, where each layer reads data from a subset of the memory bank and writes or overwrites the result back into another subset of the bank. This approach enables the exploration of a broad range of architectures with variable depth, connectivity patterns and layer sizes. Elsken et al. [75] apply network morphisms to generate large *NN* architectures that consider skip connections and yield to competitive performances on established data sets and to accelerate the convergence of new neural architectures further.

Inspired by hand-crafted architectures that often consist of repeating layers, recent developments in *NAS* [225, 107, 115] consider a block-wise search space, where blocks of layers are parametrised and concatenated.

Zoph et al. [256] propose the *NASNet* search space. In *NASNet* different convolutional blocks are configured and stacked to a *CNN* architecture. By stacking blocks, (i) the size of the search space is reduced while achieving better performances, (ii) build architectures can easily be transferred to other data sets or included in other architectures, and (iii) the repetition of patterns in previous architectures is mapped to *NAS*. The smaller search space while achieving better performances [198] has led to the adaption of block-wise architectures [198, 188, 73, 45]. Building architectures from blocks, new design choices arise. While Zoph and Le [255] build a sequential block-wise *CNN* architecture, Cai et al. [45] employ a meta architecture, where blocks can be combined arbitrarily [74]. Splitting the global search space into a micro search space (configurations of blocks) and a macro search space (links between the blocks) simplifies the search process but should be carried out simultaneously. The inner structure of the block influences the design choice of the meta-structure, and a mishandled meta-structure leads to non-performing architectures about the validation protocol $\mathcal{V}$ and the loss $\mathcal{L}$.

However, the choice of the search space $G, \mathcal{Z}, \Lambda$ indicates the difficulty of the optimisation problem. Zoph and Le [255] e.g. explore their search space by utilising 800 *GPU* for 28 days. The difficulty of evaluating large search spaces with limited computational resources led to the development of benchmark search spaces, where the search space is commonly fully explored. NAS-*Bench-101* [247] was the first benchmark data set that contained $423k$ different architectures trained on the *CIFAR*-10 [137] for $4, 12, 36$ and 108 epochs trained for 900 *TPU* (*Tensor Processing Unit*) days. Dong and Yang [68] extended NAS-*Bench-101* by adding learning curves based on the *CIFAR*-10, *CIFAR*-100 [137] and *ImageNet* [62] data sets. However, NAS-*Bench-201* explores a smaller search space that contains $6k$ different. By building surrogate architectures Siems et al. propose NAS-*Bench-301*. It consists of 10^{18} architectures based on a *DARTS* [153] search space whereby $60k$ architectures were trained for 30 epochs. In order to build one interface for several NAS-*Bench* data sets the *NATS-Bench* [67] framework was developed Further, NAS-*Bench-NLP* [130] was developed to perform *NAS* on *NLP* tasks.

Since more complex architectures tend to perform better and already small search spaces pose the problem of remaining non-continuous, an efficient search strategy becomes necessary. In the following, we depict commonly applied search strategies in *NAS*.

2.3.3.2 Search Strategy

In the previous section, we depicted the variability of architectures and, thus, the variability and complexity of different search spaces that led to the development of benchmark data sets. In this section, we depict different search strategies that have been applied in *NAS*. According to *AutoML* solvers, *random search* [147, 164], *BO* [19, 64, 125, 252], and *EA* [155, 84, 222, 198, 229, 75, 220] methods are also applied in *NAS*. However, additionally *RL* [55, 255, 256, 226] approaches as well as gradient-based methods are commonly applied as search strategy [74]. In the following, we briefly present these different search strategies:

Trivial Search Strategies are generally applied to construct benchmark data sets, such as *NAS*-Bench [164], or to explore the entire search space with respect to the loss function [147]. While *h2o* applies *grid search* to perform *NAS* based on smaller *MLP* architectures, *random search* is often applied as baseline for new search strategies [198]. However, as the number of evaluations in *NAS* is crucial, trivial strategies require an excessive amount of evaluations and remain therefore unsuitable for *NAS*.

Evolutionary Algorithm methods have been frequently used for searching not only the optimal architecture but also searching for optimal weights of neural networks [155]. Optimizing simultaneously the architecture as well as the weights of a *NN* is also termed as *neuroevolution*

[84]. As discussed in Section 2.3.1.2, the success of *EA* in *HPO* depends on the encoding of the individuals within the population. In *NAS*, these encoding spaces are configured by the search space whether they encode the layers, blocks or the topology in a fixed, partial fixed or relatively free architecture encoding [155]. Furthermore, to enable *NAS* with *EA* approaches the (i) initialisation, (ii) selection, (iii) recombination, (iv) mutation, and (v) evaluation steps need to be specified as illustrated in Figure 2.8.

The population of *NN*s can be initialised by three types of methods: (i) trivial, (ii) random, and (iii) rich initialisation. The trivial initialisation starts with a relatively simple architecture for each individual within the population to explore and exploit more complex architectures within the evolutions. Random initialisation [221, 229] methods initialise a population by randomly sampling neural architectures and the rich initialisation [256] uses the knowledge of already well-performing models to obtain a population at the beginning of the evolution process. For the evaluation of the population, simple *NN*s with fewer parameters have the advantage that they are, compared to more complex architectures, computational less expensive to train. A trivial initialisation [198, 245, 197] can thus evolve to more and more complex and performant architectures until a stopping criteria is fulfilled. However, a rich initialisation can potentially obtain already well-performing architectures at the beginning of the search process.

The selection the strategies can be grouped as presented in Section 2.3.1.2 into (i) *roulette wheel* [245], (ii) *rank* [198], (iii) *steady state* [73], (iv) *tournament* [221] and *elitism* [229, 75, 220] selection. Real et al. [198] apply an ageing evolution, that discards the oldest individual in the population.

The existing selection strategies focus on preserving suitable architectures within the population by simultaneously fostering the diversity of the population for a better exploration of the search space. For the recombination step Sun et al. [222] proposed crossover recombination for *NAS*. The crossover operation achieved better performances against approaches without recombination in this method. However, the crossover operation also showed that it could generate offsprings dramatically different from the parents. Approaches, that skip recombination and simply mutate a parent architecture [198, 255] let an individual explore the neighbouring region. The evaluation step is the most time-consuming step in *NAS*. To reduce the evaluation time in *NAS* weight inheritance [198, 75] or an early stopping policy [222] is commonly applied in *EA* approaches. *One-shot* approaches, as proposed by Zoph et al. [256], train a single over-parametrised network, the one-shot model, where the structure contains subgraphs for other candidates of the search space. This one-shot model is used to share the weights, warm start other parametrisations, and thus reduce the required computational resources to explore a search space. Real et al. [197] propose an ageing evolution (also denoted as regularized evolution) algorithm and compare this search strategy against the *RL* approach proposed by Zoph et al. [256] and a *random search* strategy based on the *NASNet* search space. Furthermore, *TPOT* [177] is as presented in Section 2.3.2 able to perform *NAS* based on a *EA* search strategy and a *MLP* search space.

Baysian Optimization As in *AutoML*, *BO* has also been employed in *NAS* [209, 240] but the large and non-continuous search spaces of *NAS* as discussed in Section 2.3.1.3 tend to make *BO* unsuitable for *NAS*. Despite the search space, Bergstra et al. [20] created state-of-the-art *CNN* architectures by applying *BO*. Furthermore, Domhan et al. [64] achieved state-of-the-art performances by applying the *SMAC* algorithm. Motivated by the observation that when neural architectures are created by hand, humans can quickly detect that a *NN* performs poorly and terminate the corresponding training; Domhan et al. [64] mimics this early termination of poorly performing training using a *Bayesian* surrogate model that extrapolates the performance of a given architecture at the beginning of the learning process. *Auto-Keras* [125] is accordingly to the *AutoML* frameworks a *NAS* framework based on *Keras* [52] that implements *BO* as a search strategy and develops a kernel and a tree-structured acquisition (*SMAC* [117]) function to efficiently explore the search space. Another framework developed by Zimmer et al. [252] that applies an adapted *BO* (*BOHB*) technique is *Auto-Pytorch*.

Reinforcement Learning An approach not commonly applied in *AutoML* but in *NAS* is *RL* (*Reinforcement Learning*). Excessive research work to solve the *NAS* problem with *RL* based approaches [255, 11, 256] and heterogeneous search spaces has been applied. *RL* approaches generate architectures from an agent's actions, where the action space of the agent refers to the search space $G, \mathcal{Z}, \Lambda$. The agent's policy that leads to the actions is trained on an estimate of the performance of a model $f_{g, \overrightarrow{z}, \overrightarrow{\lambda}}$ (also referred to as reward) configured by the agent. This reward can be represented by the validation protocol $\mathcal{V}$ and a given loss metric $\mathcal{L}$. *RL* approaches for *NAS* mainly differ in the agent's learning model that learns a policy by optimising the reward obtained by evaluating the given neural architectures and how the agent's model is updated within the search procedure. Zoph and Le [255] use a *RNN* to sequentially encode an architecture and optimise the agent by applying a policy gradient algorithm [255]. In a later approach Zoph et al. [256] apply a proximal policy optimization. Baker et al. [11] propose a q-learning approach, where the agent interacts with the environment by sequentially building *CNN* architectures. Other *RL* approaches such as [255, 188, 226] use Policy Gradient algorithms.

For practical usability, frameworks have emerged around *NAS* procedures or that have been explicitly developed for *NAS*. In Table 2.3, we provide an overview of the most common *NAS* frameworks. Contrary to *AutoML* frameworks for *NAS* differ more in the goal they pursue. In order to include *NAS* into established *AutoML* frameworks, *TPOT* and h_2o use a *MLP* architecture as underlying search space with the goal to include the configuration of *NN* into the search space. This enables to combine the search for suited *ML* pipelines and *DNN* search spaces but possibly neglects more suitable architectures such as *CNN* and *RNN*. While *TPOT* uses *PyTorch* [184] h_2o uses *Tensorflow* [1] as underlying *DL* framework.

Table 2.3: Comparison of different *NAS* frameworks.

Framework	Search Strategy	Search Space	Benchmark
TPOT [177]	*GP*	*MLP*	×
h_2o [147]	*Grid Search*	*MLP*	×
Auto-Keras [125]	*SMAC* [117]	*Dense*, *CNN* and *ResNet* blocks	×
Auto-Pytorch [252]	*BOHB* [78]	*ResNet* and *MLP*	×
NASLib [164]	DARTS, Random Search, Bananas	*NAS*-Bench[2]	✓
Retiarii [250]	*EA*	Variable	~
Keras-Tuner	*Random Search*	Variable	~

Further frameworks depicted in Table 2.3 are *Auto-Keras* [125], *Auto-Pytorch* [252], *NASLib* [164] and *Retiarii* [250]. These frameworks differ in the goal they pursue in that *Auto-Keras* and *Auto-Pytorch* are frameworks that aim to find suitable neural architectures based on new image, text, or tabular data sets a user provides and wants to have processed. In *Auto-Keras* the underlying *HPO* (*SMAC* [117]) searches within fully connected (*Dense*), *CNN* and *ResNet* blocks for a suitable model $f_{g,\vec{z},\vec{\lambda}}$. The generated blocks are surrounded by input and output blocks to maintain compatibility with other data sets and formats. As the name of the framework suggests the underlying *DL* library is *Keras* [52] and implements *BO* as *HPO* technique. *Auto-PyTorch* [252] relies on PyTorch [184] and stacks based on *BOHB* [78] various fully connected and *ResNet* blocks. In comparison to the other frameworks *Auto-PyTorch* provides a smaller variety of neural structures but concatenates further preprocessing steps and thus bridges a *ML* pipeline and *NAS* for tabular data sets. *Retiarii* [250] as well as *Keras-Tuner* [178] aim to tune already predefined models by the user. While *Retiarii* relies on *PyTorch* and a mutation based *EA* search strategy, *Keras-Tuner* uses the *Keras* library to optimise a architecture by a *Random Search* strategy. Both *Retiarii* and *Keras-Tuner* pursue the goal of tuning already existing architectures. The search space in *Retiarii* is given by possible mutations in each layer of the architecture. In *Keras-Tuner*, the search space is specified by the user prior to the optimisation process.

Summary

To automate a *ML* pipeline, we defined in this section the automation of the search for suitable *ML* pipelines and neural architectures. We depicted in Section 2.3.1 optimisation strategies that aim to heuristically apply *HPO* on different search spaces. In order to automate the search for suitable *ML* pipelines, we formalised the pipeline creation problem and highlighted the differences to the *CASH* problem defined in Section 2.3.2. Section 2.3.2 further highlights the tools and frameworks that aim to optimise *ML* pipelines based on different underlying solvers and pipeline structures. These frameworks were compared based on their characteristics, e.g. whether they support ensembles, parallel and time constraint optimisation, or if they support *NAS* techniques. Building on *AutoML* we present in Section 2.3.3 *NAS* as an optimization problem with large and distinct search spaces $G, \mathcal{Z}, \Lambda$ and common but from *AutoML* different search strategies to optimise neural architectures.

[2] including *NAS*-Bench-101 [247],*NAS*-Bench-201 [68], *NAS*-Bench-301 [213], ASR [163] and NLP [164]

2.4 Learning to Rank

In order to enable a utility driven *AutoML* or *NAS* framework, we introduce in this section *LTR* (*Learning To Rank*) techniques. Thus, within this section, we will first take a step back from *AutoML* and *NAS* and introduce *LTR* by its initial aim, namely to rank documents. In the course of this work, the ranking of documents will be applied to the problem of ranking and thus ordering *ML* pipelines and *NN*s towards an underlying utility. First, we formalise *LTR* as a general optimisation task in Definition 11 and then depict different possibilities to solve this problem. The motivation for *LTR* techniques lies within the overwhelming flood of information; *LTR*'s aim is to construct a ranking model that can rank this information according to its relevance. Especially in *information retrieval* and *information filtering* i.ex. document retrieval, collaborative filtering, product ratings, sorting objects based on certain factors poses a central problem. A well-known approach to ranking documents is *PageRank* developed by Page [182] (Google) for ranking websites accordingly to their relevance. It works by counting the number and the quality of links to a document to determine an estimate of how relevant a document (website) is. Thereby Page [182] assumes that relevant documents are likely to have more links from other websites or documents. However, *LTR* can be seen as both a supervised and unsupervised learning problem. As Page[182] does not consider any response but is based on a graph including an adjacency matrix that represents the links connecting the different websites (documents), this approach can be considered as an unsupervised learning approach. Ranking problems can, thus, be systematically categorised by (i) the available data and by the (ii) type of response. The available data, at the top level, categorises the ranking problem into *label ranking*, *object ranking* and *instance ranking* [51]. In *label ranking* the training data consists of features $\overrightarrow{x} \in X$ and the task is to learn a Permutation $\pi(\overrightarrow{x}) \in Perm(1:t)$ [239] where

$$Perm(1:d) := \{\pi | \pi \text{ is a permutation of } \{1, \dots, t\}\}. \tag{2.18}$$

The permutation $\pi(\overrightarrow{x})$ can be interpreted in that $(\pi(\overrightarrow{x}))_1$ denotes the most preferred document. In *instance ranking* the training data considers a tuple $(\overrightarrow{x}, y) \in \mathcal{D}$, where $\overrightarrow{x} \in X$ represents the features and $y \in Y$ the ordering of a document. In *object ranking* the goal is to learn a ranking function that produces a ranking based only on the features $\overrightarrow{x}$ of a document. Based on *object ranking* Cheng [51], we define the general *LTR* problem as follows:

> ➢ **Definition 11.** Learning to Rank, adapted from Werner [239]:
> Consider $\mathcal{D}$ as set of t documents $\mathcal{D} = \{\overrightarrow{x}_1, \dots, \overrightarrow{x}_t\}$, where each document is represented by a vector $\overrightarrow{x} \in X$. The documents in $\mathcal{D}$ can by ordered in that $\overrightarrow{x}_i \succ \overrightarrow{x}_j$ and the goal is to find a ranking function f that assumes as input a set of documents and returns a permutation of this set.

Based on *object ranking* defined in Definition 11, we assume an *instance ranking* setting where in order to develop a supervised learning task (see Definition 1) labels are available.

Ranking problems can further be characterised by the type of the response [239]. These responses can be a binary indication of whether a ranked document is relevant or not, ordinal for this indication of an index $\{1, \dots, t\}$ or continuous for a scoring based ranking function. A binary or ordinal indication of relevance can be a binary or multi-class classification problem. The number of classes within the multi-class case corresponds to the number of documents to rank. Considering a continuous measurement for the relevance of documents has the advantage that an ordinal ranking can be derived. With a given threshold of relevance,

a binary indication can be modelled. We assume a continuous measurement for relevance scoring in the following, which maps to a regression problem.

LTR can further be categorised into pointwise [58, 151] , pairwise [102, 44, 176, 207] and list-wise [46, 120] approaches. In the following, we derive, based on Definition 11, the pointwise, pairwise and list-wise indications of preferences and provide common approaches.

2.4.1 Pointwise

The pointwise approach assumes that each training document or observation with features $\overrightarrow{x}$ is associated with a ranking measurement. Thus the problem can be reduced to a regression problem. However, considering a pointwise indication of a relevance score, the question of a suitable data collection method arises. Pointwise approaches assume that one can rate documents with absolute ratings. To train such a ranking model, a user has to be aware of all documents in $\mathcal{D}_{train}$ to construct a suitable data set that assigns the correct relevance score to each document. To train the pointwise ranking problem, it can be formalised as follows:

> ➢ **Definition 12.** Pointwise Ranking, based in Definition 1:
> Let $\mathcal{D} = \{(\overrightarrow{x}_1, y_1), \ldots, (\overrightarrow{x}_t, y_t)\}$ be a set of documents, where each document has a label $y \in \mathbb{R}$, then the task is to learn a function $f^* : X \to \mathbb{R}$ (also referred to as ranking model), that transforms a document by it's feature vector $\overrightarrow{x} \in X$ into a relevance score $y \in \mathbb{R}$. Furthermore, let $\mathcal{L}(f(\overrightarrow{x}), y)$ be a loss function that quantifies the correctness of a series of predictions of f trained on $\mathcal{D}_{train} \subset \mathcal{D}$ for $\overrightarrow{x}$. Denote that $\{\overrightarrow{x}, y\} \notin \mathcal{D}_{train}$ and $\mathcal{D}_{train} \cap \mathcal{D}_{valid} = \emptyset$. Then the goal to is similar to Equation 2.1 to minimise the loss $\mathcal{L}$ on $\mathcal{D}_{valid}$ using a validation protocol $\mathcal{V}(\cdot, \cdot, \cdot, \cdot)$.

In Definition 12 we define the *pointwise ranking* problem as a supervised regression problem, where the task is to learn a ranking model $f : X \to \mathbb{R}$ that predicts from a feature vector $\overrightarrow{x}$ of a document a relevance score. A number of existing supervised *ML* algorithms (see Section 2.2.4) can be readily used for this purpose. Prominent examples are PRank [58], which performs the ranking by an ordinal regression, and McRank [151], which employs multi-class classification and Gradient boosting techniques. However, pointwise approaches are not advantageous for our problem setting, as it is difficult for end-users to assign an absolute target value for the suitability of a machine learning model.

2.4.2 Pairwise

Pairwise ranking approaches are trained based on pairwise ranked documents. Based on this information, a ranking model learns a global ranking for all documents in $\mathcal{D}$. The model is trained based on a document tuple with the corresponding label $(\overrightarrow{x}_i, \overrightarrow{x}_j, y)$. The label y states the order of both documents in that $y \in \{-1, 0, 1\}$ and thus if a document i with its corresponding features $\overrightarrow{x}_i$ is more, equal or less relevant than document j with features $\overrightarrow{x}_j$. Again, pairwise ranking models can be characterised by their response type independent of training. The response type of common models is either following the pairwise training process, the comparison of two documents (i.ex. $\overrightarrow{x}_i \succ \overrightarrow{x}_j$), or a relevance scoring that enables the ranking of documents in $\mathcal{D}$. Considering a comparison based response, the model's input requires two documents. In contrast, the relevance scoring method (see Definition 12) only requires one document to

predict a document's relevance score. In Definition 13, we define the pairwise ranking problem based on pairwise comparisons.

➢ **Definition 13.** Pairwise Ranking
Let $\{\overrightarrow{x}_1, \ldots, \overrightarrow{x}_t\}$ again be a set of documents, where each document is characterised by $\overrightarrow{x} \in X$. Further, let $\mathcal{D} = \{(\overrightarrow{x}_1, \overrightarrow{x}_1, y_1), \ldots, (\overrightarrow{x}_t, \overrightarrow{x}_t, y_t)\} \in X \times X \times \{-1, 0, 1\}$ be a set of document pairs labelled by $y \in \{-1, 0, 1\}$. The label y indicates whether document i is more relevant than document j ($\overrightarrow{x}_i \succ \overrightarrow{x}_j$) vice versa ($\overrightarrow{x}_i \prec \overrightarrow{x}_j$) or ranked equally as favourable. Based on a training data set D_{train} the task is to learn a function $f^* : X \times X \rightarrow \{-1, 0, 1\}$. Furthermore, let $\mathcal{L}(f(\overrightarrow{x}_i, \overrightarrow{x}_j), y)$ be a loss function that quantifies the correctness of a series of predictions of f trained on $\mathcal{D}_{train} \subset \mathcal{D}$. Denote that $(\overrightarrow{x}_i, \overrightarrow{x}_j, y) \notin \mathcal{D}_{train}$ and $\mathcal{D}_{train} \cap \mathcal{D}_{valid} = \emptyset$. Then the goal to is similar to Equation 2.1 to minimise the loss $\mathcal{L}$ on $\mathcal{D}_{valid}$ using a validation protocol $\mathcal{V}(\cdot, \cdot, \cdot, \cdot)$.

Considering that the ranking models response is a ranking score $f(\overrightarrow{x}) \in \mathbb{R}$ predicted based on one document the loss of the trained model can be modelled by a surrogate function

$$g(\overrightarrow{x}_i, \overrightarrow{x}_j) = \begin{cases} 1 & , \quad f(\overrightarrow{x}_i) > f(\overrightarrow{x}_j) \\ 0 & , \quad f(\overrightarrow{x}_i) = f(\overrightarrow{x}_j) \\ -1 & , \quad f(\overrightarrow{x}_i) < f(\overrightarrow{x}_j) \end{cases} \tag{2.19}$$

that is then used to calculate the loss $\mathcal{L}(g(\overrightarrow{x}_i, \overrightarrow{x}_j), y)$ based on a previously defined ranking y. Common approaches for *pairwise ranking* are Ranking SVM [102], RankNet [44], LambdaRank [43] and LambdaMART [89]. Ranking SVM [102] uses a support vector machine to maximise the span between the classes correctly and incorrectly ranked to solve the pairwise ranking problem. RankNet [44] uses *NN*s to, on the one hand, learn a relevance score $f(\overleftarrow{x})$ for both documents in a pair within a Siamese architecture and, on the other hand, to maximise the cross-entropy between the predicted relevancies ($g(\overrightarrow{x}_i, \overrightarrow{x}_j)$). LambdaRank [43] extends RankNet by not using the actual costs (number of inversions in ranking) but their gradients for training. Lastly, LambdaMART [89] combines LambdaRank and multiple additive regression trees. In comparison to pointwise approaches, pairwise ranking approaches have the advantage that the training data set $\mathcal{D}_{train}$ can be generated without global knowledge about other documents to indicate a relevance score.

2.4.3 Listwise

Listwise ranking approaches learn to rank documents by partial lists of documents that are already ordered. Similar to the *pairwise ranking* approach, the output of *listwise ranking* models can either be a relevance score for each document ($\overrightarrow{y}$) whereby the documents are ordered separately based on the predicted relevance score or the output of the model can directly be an ordering for the selected documents i.ex. a tuple with the document indices, where the tuple is sorted based on their predicted relevance. In Definition 14, we define the *listwise ranking* problem.

➢ **Definition 14.** Listwise Ranking
Let $\{\overrightarrow{x}_1, \ldots, \overrightarrow{x}_t\}$ again be a set of documents, where each document is characterised by $\overrightarrow{x} \in X$. Further, let $\mathcal{D} = \{(\overrightarrow{x}_k, \ldots, \overrightarrow{x}_l,), \ldots, (\overrightarrow{x}_i, \ldots, \overrightarrow{x}_j)\}$ be a set of sorted document tuples of length n, whereby $(\overrightarrow{x}_i, \ldots, \overrightarrow{x}_j) \in X \; \forall i, j \leq t$. Based on $\mathcal{D}$ the task is to learn a function $f^* : X^n \rightarrow Y$ that indicates the ranks $\overrightarrow{y} \in Y$ of unsorted tuples $\{(\overrightarrow{x}_i, \ldots, \overrightarrow{x}_j)\}$. Furthermore, let $\mathcal{L}(f((\overrightarrow{x}_i, \ldots, \overrightarrow{x}_j)), \overrightarrow{y})$ be a loss function that quantifies the correctness of a series of predictions of f trained on $\mathcal{D}_{train} \subset \mathcal{D}$. Denote that $(\overrightarrow{x}_i, \ldots, \overrightarrow{x}_j) \notin \mathcal{D}_{train}$ and $\mathcal{D}_{train} \cap \mathcal{D}_{valid} = \emptyset$. Then the goal is to similar to Equation 2.1 to minimise the loss $\mathcal{L}$ on $\mathcal{D}_{valid}$ using a validation protocol $\mathcal{V}(\cdot, \cdot, \cdot, \cdot)$.

To train the *listwise ranking* approach, Cao et al. [47] propose in *ListNet* a loss function that enables to train *NN*s based on listwise rankings of documents. In *NDCG* Valizadegan et al. [232] approach this problem by breaking the list of documents down into pairwise comparisons and optimising the normalised discounted cumulative gain. Ibrahim and Landa-Silva [120] propose in *ES-Rank* a *ES* based approach to rank documents based on seven different evaluation metrics. In *SetRank*, Pang et al. [183] learn based on the already for the training process ranked lists an encoding function that encodes and ranks lists of documents of any size. As stated in [232] the *listwise ranking* approach can be reduced to a *pairwise ranking* approach. Consider a ranked and thus sorted list of documents i.ex $(\overrightarrow{x}_3, \overrightarrow{x}_1, \overrightarrow{x}_4)$ where the most relevant document is represented by $\overrightarrow{x}_3$ and the least relevant document by $\overrightarrow{x}_4$, then this list can be reduced to a set of pairwise comparisons $\{(\overrightarrow{x}_3, \overrightarrow{x}_3, 0), (\overrightarrow{x}_3, \overrightarrow{x}_1, 1), \ldots\}$. This enables to apply approaches presented in Section 2.4.2 for the *listwise ranking* scenario.

Summary

In this section, we introduced *pointwise*, *pairwise*, and *listwise* ranking approaches that enable the ranking documents and to obtain their relevance compared to other documents. We presented the *pointwise ranking* approach that can be reduced to a supervised learning problem depicted in Definition 1 and the *pairwise ranking* approach, where the task is to learn absolute rankings (or relevance scores) based on pairwise comparisons. The *listwise ranking* approach, where the ranking model is trained based on sublists to predict a global ranking of documents, can be reduced to a *pairwise ranking* scenario by splitting sorted sublists into pairwise comparisons of documents. For the moment, the added utility of ranking models in the context of *AutoML* remains unclear; however, in the course of this work, we will replace the ranking of documents by the ranking of *ML* pipelines generated by *AutoML* instances and neural architectures generated by *NAS* instances. The connection of ranking algorithms to *HPO* in *NAS* and *AutoML* is addressed and established in part III.

2.5 Learning on Data Streams

The purpose of this section is to provide the foundations for *ML* algorithms that are able to adapt to changes in the underlying data patterns. The current literature in the field of *ML* focuses its primary emphasis on algorithms that can learn from large amounts of data that are already available at the beginning of the learning process. Within the *DM* process in Section 2.1, we motivated a traditional, iterative approach towards a *ML* pipeline trained on a given data batch. *Online learning*, however, implies that we are no longer referring to data sets $\mathcal{D}$ that are already available, but rather to data streams $\mathcal{S}$ on which an *ML* algorithm adapts

incrementally on continuously incoming data instances. As this section moves from batch learning to an *online learning* setting, the differences in terminology are illustrated below according to Frochte [90]:

Online learning: Learning a *ML* model based on data streams without storing the data.

Offline learning: The entire data set $\mathcal{D}$ is known from the beginning and can be retrieved at any time.

Incremental learning: Learning of individual data points (instances) one by one.

Batch learning: The model is trained based on subsets of a data set $\mathcal{D}$.

It becomes clear that in an *offline learning* scenario, a *ML* model can be trained with a batch of data points as well as incrementally. Denote that the order of the data points within a given data set $\mathcal{D}$ can also have significant relevance in *offline learning*. The setting where the underlying data points' ordering is relevant is also referred to as a univariate or multivariate time series. However, in an *online learning* scenario, the boundary of storing no data is often blurred, as the storage of mini-batches is often considered as an *online learning* method [23]. In Figure 2.13, we illustrate based on the *ML* pipeline and the *DM* process the transition from an offline learning approach developed within a *DM* process under the assumption of evolving data streams.

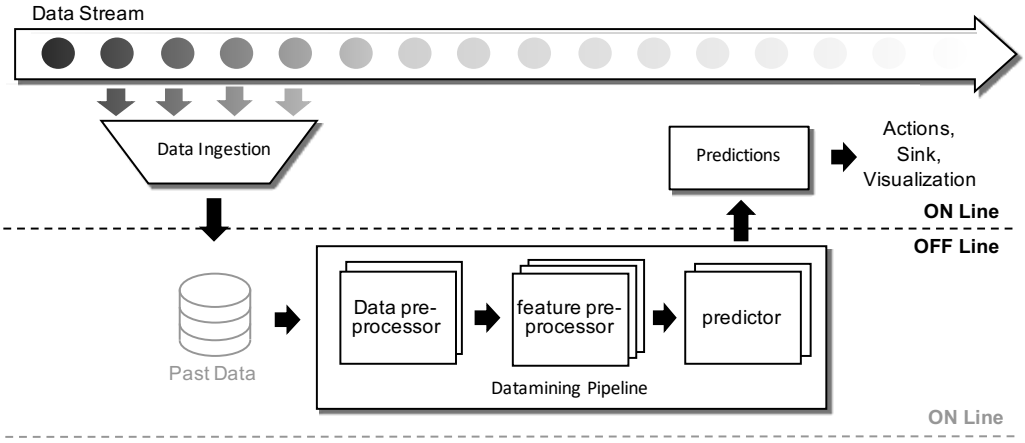


Figure 2.13: From Batch to Online Learning: An illustration for adaptive learning on data streams.

When considering ever-evolving data streams within a *DM* process, it remains clear that the generation of data and the application of the model are integrated into the actual process, while the development of the model takes place separately, starting from an isolated data set. As Figure 2.13 shows, the development and thus the learning of a *DM* pipeline takes place outside (offline) of the incoming data points. Further, it is evident that in the meantime, new data is already being generated, which had not been taken into consideration during the offline learning process but which could also contain relevant information or changes in the underlying data distribution can occur. This inability to iteratively integrate new and more recent data into an existing model is probably the most critical drawback of offline learning approaches. Instead, the generation of a new data set within the data ingestion step is required, and the already developed *DM* pipeline might be inappropriate for the newly generated data set and thus for the problem at hand. Furthermore, the larger and higher-dimensional the data set, the more critical the scalability and efficiency of offline algorithms in terms of time and memory [112]. However, this does not concern the predictive performance but also the training process. Since the training process is computationally more expensive than the model usage, the exploitation of computer resources is concentrated during offline learning at a time when all data points from the generated data set are processed. Concerning Figure 2.13, it would be desirable to skip the data

ingestion step and to design a system where no separation between the evolving data stream and the model development is made and thus where new data is immediately processed by a defined *ML* pipeline. To enable the immediate processing of the incoming data stream, we formalise in Section 2.5.1 the *online learning* problem, depict the requirements for an online learning system and in Section 2.5.2 we introduce the change of data patterns and distributions within the data stream, which is referred to as *concept drift*. According to the steps of an offline *ML* pipeline presented in Section 2.2, we depict in Sections 2.5.3 to 2.5.4 the steps required to build online *ML* pipelines. Since online ensemble methods are closely related to online *AutoML* in that they train sets of homogeneous or heterogeneous models to perform a prediction on changing data streams, we discuss online methods as part of related work for online *AutoML* in Section 3.4.

2.5.1 Online Learning

Advancing an adaptive *ML* framework, it becomes clear that it is crucial to have algorithms at hand that can process data that arrives continuously in the form of data streams. However, online analysis brings challenges and aspects that need to be considered. First, *online learning* has to deal with potentially real-time data rather than previously known data sets. We refer to an incremental online learning system that enables the adaptation to data streams, which leads to some requirements for the learning process. This includes differences from offline learning in evaluating and monitoring the model's performance, which is addressed in Section 2.6. Further, Bifet et al. [23] depict the requirements for online learning as follows:

Requirements I. Online Analysis Algorithms, adopted from Bifet et al. [23]:

R I–1. Process an instance at a time, and inspect it (at most) once.

R I–2. Use a limited amount of time to process each instance.

R I–3. Use a limited amount of memory.

R I–4. Be ready to give an answer (e.g. prediction) at any time.

R I–5. Adapt to temporal changes.

Since data streams are potentially infinite and new observations may arrive with a high frequency, stream algorithms should be efficient in terms of resource usage, i.e. time (see *R I–2*) and memory (see *R I–3*) consumption. The Requirement *R I–1* stems directly from the main characterisation of data streams and describes the nature of *incremental learning*. Data points might come one at a time and in a sequence, so they must also be processed in this order by a corresponding streaming algorithm. Each instance is considered exactly once and then discarded. Therefore, no storage takes place, whereby this restriction is blurred by approaches that consider mini-batches and that are also referred to *online learning* approaches. Including a model into a real-time environment, Requirement *R I–2* and *R I–3* become relevant, but above all the algorithm must also be capable to give a response at any time (Requirement *R I–4*). This can be achieved when the complexity at runtime of the algorithm is linear to the number of data points processed. When the frequency of the incoming data points exceeds the time required to process a data instance from the stream, the process over time potentially leads to data loss and unavailability of the model. Thus, *R I–4* gets infringed. A significant difference compared to offline learning comes with Requirement *R I–5*. While in *offline learning*, there is a clear distinction between the training phase and the prediction phase, in *online learning*, the model handles evolving data streams that may never end. Thus a separation between the training and the prediction phase is not possible. Furthermore, the underlying data distribution or concept

might change over time. Changes in the data concept over time are also referred to as *concept drift*. However, Requirement *R I–5* refers to the capability of a model to adapt to data streams.

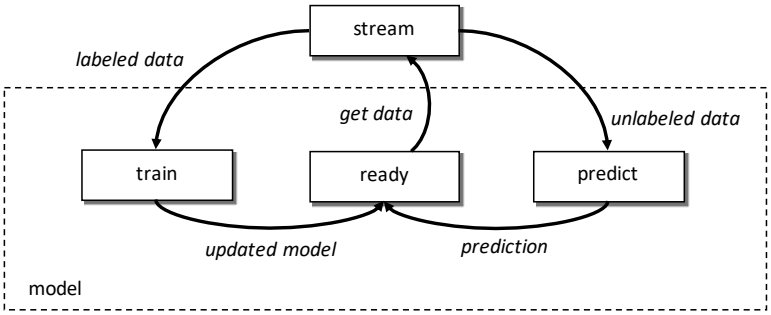


Figure 2.14: Illustration of an online learning system, following [172]

Figure 2.14 exemplary shows how an online learning framework is able to comply with the stream requirements depicted within the Requirement I [23] for a supervised learning task depicted in 1. If the data point in the data streams is a labelled data point $(\overrightarrow{x}, y)$, then the model processes the instance by updating the model. When the data point is an unlabelled instance $\overrightarrow{x}$, the task is to predict a label $\hat{y}$. Thus, the model processes each instance from an evolving data stream, updates the underlying model and is ready to predict at any time.

In Definition 15 we define, following the supervised offline learning problem in Definition 1 the *online learning* problem.

➢ **Definition 15.** Supervised Online Learning
Let $\mathcal{S} = e_1, \ldots, e_t, \ldots$ be an ordered sequence of examples of possibly infinite length and let e_t be the current observed examples. Each example e_i (e.g. $e_i = (x_i, y_i)$) is a tuple of p predictive attributes $x_i = (x_{i,1}, \ldots, x_{i,p})$ and in the supervised learning setting a label y_i. Further, let $\mathcal{S}^- = e_0, \ldots, e_t$ be an ordered sequence of past examples. Then the task is to learn a function $f^* : X \rightarrow Y$ (also referred to as model) that minimises the loss $\mathcal{L}$, based on training $\mathcal{S}_{train} \subseteq \mathcal{S}^-$ and validation $\mathcal{S}_{valid} \subseteq \mathcal{S}^-$ samples. Denote that $\mathcal{S}_{train} \cap \mathcal{S}_{valid}$.

$$A^* = \min \mathcal{V}(\mathcal{L}, f, \mathcal{S}_{train}, \mathcal{S}_{valid}) \tag{2.20}$$

Instead of referring to a set $\mathcal{D}$ of labelled data points $(\overrightarrow{x}, y)$, we refer in Definition 15 to an ordered sequence $\mathcal{S}$ of data instances $e_i = (\overrightarrow{x}, y)$. Furthermore, it differentiates between past $\mathcal{S}^-$ and future instances, whereby the model is validated and trained on past data. In the following, we discuss the arbitrary of changes in evolving data streams on which a model f should react and adapt.

2.5.2 Concept Drift

Changes in the pattern of the data stream over time are also referred to as *concept drift* [242]. Shifts of the underlying distribution or pattern occur especially in dynamically changing and non-stationary environments, e.g. the change in users' interests when following an online news stream [92]. However, *concept drifts* can

occur in the form of changes within the characteristic of the input features $\overrightarrow{x}$, but also the relation between the input data and the target y. Gama et al. [92] define this change by changes in (i) the prior probabilities for a label y $p(y)$, (ii) the target probability with regard to the features $p(\overrightarrow{x}|y)$ or the posterior probabilities of the target value $p(y|\overrightarrow{x})$ that may change. This shift of the underlying distribution of the data stream can therefore proceed in different forms that are depicted in the following [23]:

Sudden concept drifts occur when the distribution or pattern changes in a few steps of the data stream. It is also referred to as shift. A model should, in this case, detect the shift and adapt as rapidly as possible to this shift. For instance, sudden concept drifts may occur when the mode of operation of an *IIoT* or *IoT* sensor changes due to external influences such as movements.

Gradual changes are shifts that occur incrementally over a more extended period. Therefore, the model's challenge is to adapt to the continuously changing data stream. An example is the abrasion of production machines measured by various sensors.

Partial concept drifts might affect only instances of certain forms i.ex. the change of the data stream concerns a few classes in a classification task, or only a part of the features processed by the model f are affected by the shift that is either sudden or gradual.

Recurrent concepts are changing distributions that have already appeared in the past and tend to reappear later in the stream. The challenge of a *ML* model is, therefore, to remember the different data patterns and to select the proper function that transforms the features into the target value.

In the context of *concept drifts*, *outlier* or *anomaly* detection algorithms are also to be mentioned and distinguished from *concept drifts*. While the change in data patterns are true changes within the data stream, *anomalies* or *outliers* are limited to very few data points that do not comply with the data pattern to be learned. Thus a change detector such as *ADWIN* [22] is commonly applied to detect changes within the data patterns [21]. *ADWIN* is a popular window based change detector, that holds and compares two sub-windows to detect changes within the data distribution. Further popular *change detectors* are based on the models error rate, such as *DDM* [91], *EDDM* [8] or on Hoeffding's bounds *HDDM* [29].

2.5.3 Preprocessing

Besides the learning process itself, there are also differences in the preprocessing of the data. As discussed in Section 2.2, in *offline learning* it is common to preprocess features $\overrightarrow{x}$ from a data set $\mathcal{D}$ at the beginning of *DM* process. While the data preparation step can be applied similar to the data preparation step depicted in Section 2.2.2. However, in *online learning*, possible errors within the data stream can be filtered or repaired by predefined rules and thus directly be applied to the data stream. Thus, it is assumed that possible errors within the data stream are known in advance in this environment. The data preprocessing, however, differs from the *offline learning* scenario since not all information of the data stream is available at the beginning of the stream. Relevant features must be defined in advance, or dimensionally reduction algorithms that automatically extract and select valuable features and that fulfil the Requirements I need to be applied. When it comes to normalisation and standardisation mechanism to enable an online *ML* pipeline, the differences to an *offline learning* approach become apparent. Normalisation is achieved by considering the smallest and largest value per feature, while standardisation involves using the mean and standard deviation of the training samples. In both cases, the data set must be known at the beginning to extract the corresponding values [90]. However, to enable normalisation and standardisation in an *online learning* environment, running statistics

can be calculated. In this case, the mean and variance are updated every time a new instance arrives and is then used to scale the features of the new data point from the data stream $\mathcal{S}$. For example the *CMA* (*Cummulative Moving Average*) $\overrightarrow{\mu}_t$ can be updated incrementally at a point in time t for all features in $\overrightarrow{x}$ as follows:

$$\overrightarrow{\mu}_t = \overrightarrow{\mu}_{t-1} + \frac{\overrightarrow{x}_t - \overrightarrow{\mu}_{t-1}}{t} \tag{2.21}$$

and thus, the *MSTD* (*Moving Standard Deviation*) $\overrightarrow{\sigma}_t$ can be calculated by applying the Welford's [238] method to estimate the variance for all features in $\overrightarrow{x}$ in an online manner:

$$\overrightarrow{\sigma}^2_t = \overrightarrow{\sigma}^2_{t-1} + (\overrightarrow{x}_t - \overrightarrow{\mu}_t) * (\overrightarrow{x}_t - \overrightarrow{\mu}_{t-1}) \tag{2.22}$$

By considering the differences between the sums of squared differences for t and $t-1$ the Welford's method enables incremental updates of the *MSTD* as well as the variance. Accordingly, the informative value of the mean and the variance increases with the rising number of instances. For selecting suitable features from a data stream, as discussed in Section 2.2.2, a drawback of *online learning* methods becomes apparent. Since the data stream evolves, the correlation and the information gain of individual features remain unclear. Here, features that might be useful are commonly selected by hand at the beginning of the data stream. The question for adaptivity within the feature selection process remains open in *online learning* and is addressed in more detail in Part IV.

2.5.4 Online Learning Models

In this section, we present the foundations for different online algorithms that are capable to minimise the loss metric $\mathcal{L}$ using a validation protocol $\mathcal{V}(\cdot,\cdot,\cdot,\cdot)$ (see Definition 15) based on the Requirements I. Further, some approaches presented in Section 2.2.4 are by nature fully comparable with the online learning requirements. By presenting different concepts of models used for *online learning*, we concentrate on the differences to *offline learning*. In Section 2.5.4.1, we depict *Decision Trees* as commonly applied classification algorithm and in Section 2.5.4.3 we present the foundations for applying *NN*'s to an *online learning* setting. Since online *NN*'s are also part of related work, we refer to Section 3.4 for a profound overview of approaches in training *NN*'s in an online manner. *NB* (*Naïve Bayes*) is also a broadly applied approach in *online learning* and is presented in Section 2.5.4.5. To conclude the foundations for online algorithms, we present in Section 2.5.4.4 Neighbourhood based approaches. Contrary to offline ensemble learning models discussed in Section 2.2.4.6 and due to the fact that ensemble techniques are closely related to *AutoML* techniques, online ensemble techniques are discussed as part of the related work in Section 3.4.

Some algorithms have been specifically created and adapted to operate on data streams. For instances, *HT* [65], *HAT* [21] or *Logistic Regression.* Other algorithms require adaption so that they can be used in an online fashion, such as creating mini-batches or introducing a sliding window [10, 180]. In the following, we depict the main concepts for *online learning* models.

2.5.4.1 Decision Tree

Following the decision tree approach presented in Section 2.2.4.1, Domingos and Hulten [65] propose an incremental, anytime decision tree assuming that the distribution of the underlying data stream does not change over time. *HT* is based on the idea that it is helpful to consider only a small subset of the training samples at a node to find the best attribute test for that node. This means that the first incoming samples of the data stream are used to determine the attribute test at the root node. Subsequent data instances are

passed through the node to different leaves according to their attribute values and used to determine the next attribute test. The whole process continues recursively, whereby *HT* uses a *Hoeffding bound* that quantifies the number of nodes to be changed to estimate the underlying distribution. While *HT* approaches are suitable for incremental learning, changes within the constructed tree structure and, thus, changes in the underlying data pattern are not considered. With *HAT* (*Hoeffding Adaptive Tree*) Bifet and Gavaldà [21] propose an adaptive approach by introducing an *ADWIN* change detector. The change detector in *HAT* is used as an indicator of whether an alternate tree should be applied to adapt to an occurring *concept drift*. Also based on *HT*, Domingos and Hulten [65] propose in addition to *HT* a *CVFDT* (*Concept-adapting Very Fast Decision Tree*), where a decision tree is kept consistent based on a sliding window, whereby alternate sub decision trees are created and replaced within the data stream when the data pattern changes at a node of the tree.

2.5.4.2 Support Vector Machine

SVM (*Support Vector Machine*) have been successfully applied in offline learning with high dimensional data sets since the generated support vectors only depend on a minimal subset of the training data. As discussed in Section 2.2.4.2, the main disadvantage of *SVM* is when training on noisy data, where a good separation of the training data by the support vector becomes difficult or impossible. Syed et al. [224] propose an approach based on mini-batches and empirically show the ability of handling *concept drifts*. Since a *SVM* builds on support vectors that consider only a small subset of the training data points, the question of how to decide whether an instance from the data stream is used as a support vector or not without the knowledge of all data within a data stream $\mathcal{S}$ arises. This question, however, is tackled by Duan et al. [69] in a separate Lagrangian optimisation problem without considering concept drifts. Regarding the Requirements I, *SVM*s are suitable for incremental learning [69] by applying mini batches but infringes Requirement *R I–1* (one instance at a time). However, the approach proposed by Syed et al. [224] does have the possibility to adapt to concept drifts. Still, it needs mini-batches to decide if new (computationally expensive) support vectors should be created.

2.5.4.3 Neural Networks

*NN*s can range from simple linear or logistic models to complex *MLP*, *CNN* and *RNN* structures with several layers (see Section 2.2.4.5) and are due to their variability in general well suited for *incremental* and *online learning*. However, not all architectures and optimisation techniques are appropriate to fulfil the Requirements I. Larger *CNN*'s such as *AlexNet* [138] or *VGGNet* [215] violate the Requirements *R I–2* (limited time) and *R I–3* (limited memory) and cannot always yield to good performances, as the deeper layers require more time to convergence. Further, the optimiser that optimises the weights of a *NN* influences the capability to be suitable for online learning. While *SGD* processes each instance at a time (Requirement *R I–1*) by nature, *Adam* or *AdaGrad* that in batch learning have prevailed for many problems, the weight updates of a *NN* is affected by the batch size as the distribution of gradients for larger batch sizes has a much heavier tail. Baydin et al. [14] propose a general online method for improving the convergence rate of gradient based optimisers used in *NN*s. The approach extends *SGD* and *Adam* by dynamically updating the learning rate during optimisation and thus improving the adaptivity of *NN*s when *concept drifts* (Requirement *R I–5*) occur.

Besides the optimiser, the choice of architecture is crucial. Linear regression or logistic regression as simplest *NN*s are already commonly used as baselines in *online learning* that are, due to their simple architecture and large throughput of instances, suitable to adapt to changes within the data stream. Most approaches applied in deeper *online learning* models are *MLP*s [123, 100]. However, the larger the model becomes, the more

difficult it is to converge and adapt to a dynamic environment. Further, they consume more memory for storing the weights and are computationally more expensive. Lobo et al. [156] propose the usage of spiking neural networks to model the behaviour and learning potential of the brain and, thus, to adapt to data streams when drifts occur. Jain et al. [123] provide an overview for *NN*s with *online learning* capabilities, that is further discussed as part of the related work in Section 3.4.

2.5.4.4 Neighbours

k-NN (*k-Nearest Neighbors*) based approaches are by nature suitable for *incremental learning*. As discussed in Section 2.2.4.3, the training is performed by storing the labelled data points to then estimate the closest relation within the feature space X on unseen data points. The fact that the basis of this approach is the memorisation of data points from the data stream raises the question about the *online learning* capabilities. Further, the data stream is possibly infinite, and thus the memory consumption of *k-NN* approaches likewise grows to an unlimited size. Especially it violates the memory Requirement ***R I–3*** as well as the ability to adapt (Requirement ***R I–5***) to *concept drifts*. To adapt *k-NN* based approaches to the Requirements I, a sliding window is introduced that, on the one hand, limits the memory consumption to the size of the window of data points and, on the other hand, enables the adaption to *concept drifts*. The adaption to *concept drifts* is ensured since the sliding window stores recent data points from the data stream and deletes the oldest data points. However, the capability of adaptation to the data stream is highly influenced by the window size as the adaptation is carried out with the delay of the size of the window and a too-small window size restricts the ability to learn the underlying data patterns within the sliding window. This windowed *k-NN* approach can further be extended with an *ADWIN* change detector to decide which samples to keep and which ones to forget. The prediction is applied similar to the *offline learning k-NN* approach, discussed in Section 2.2.4.3, by applying a majority voting in the case of classification or a uniform or weighted average in the case of regression based on k nearest neighbours.

2.5.4.5 Naïve Bayes

As discussed in Section 2.2.4.4 *NB* (*Naïve Bayes*) is a classification algorithm based on the assumption of conditional independence between the features $\overrightarrow{x}$ given the label y. In order to meet the *online learning* requirements, the estimation for the probabilities $P(x_1, \ldots, x_n|y)$ and $P(y)$ needs to be calculable and thus updated in an *online learning* manner. As stated in Equation 2.6, $P(x_1, \ldots, x_n|y)$ is estimated by the product of the conditional probabilities for each feature in $\overrightarrow{x}$ given the label y. This probability $P(x_i|y)$, as well as the class probability $P(y)$, can be incrementally updated by increasing the feature counts given the label y for each feature in $\overrightarrow{x}$. Despite the strong assumption of conditional independence for the features and thus its ineptitude for real-world applications, approaches based on Bayes are by nature suited for *online learning*. Thus extensions to this approach, e.g. *GNB* (*Gaussian Naive Bayes*) [199], augmented *NB* Zhang, or multinomial *NB* can be applied for this scenario.

Summary

In this section, we provided the foundations for *ML* algorithms that can adapt to changes in the data, also referred to as *concept drifts*. Further, we switched from an *offline learning* scenario to an *online learning* approach where we assume instead of a data set $\mathcal{D}$ a possibly infinite data stream $\mathcal{S}$. By distinguishing between *online learning*, *offline learning*, *incremental learning* and *batch learning*, and by defining the

requirements for *online learning*, we were able to examine the preprocessing and further algorithms for their suitability to an *online learning* scenario.

2.6 Evaluation Protocols

In this section, we cover the final necessities that enable a utility-based adaptation and the adaptation to data streams of *AutoML* as well as *NAS*. Within the previous sections we introduced the offline (Definition 1), as well as the online supervised learning task (Definition 15) by minimising a function f based on a metric $\mathcal{L}(\cdot, \cdot)$ and a validation protocol $\mathcal{V}(\cdot, \cdot, \cdot, \cdot)$ without giving a more detailed explanation which metrics might be considered, how a validation protocol may be implemented within the presented definitions and how the differ regarding an *online* or *offline learning* scenario. In Section 2.6.1, we present different loss metrics that can be used to train and to evaluate a singe model f, but also to steer an *AutoML* or *NAS* optimisation process. Considering an *offline learning* scenario, we present in Section 2.6.2 different evaluation protocols that evaluate a model f or a pipeline $\mathcal{P}$ on a metric $\mathcal{L}$ based on data batches $\mathcal{D}$. Finally, in Section 2.6.3, we present different from the *offline learning* scenario differing evaluation techniques assuming a data stream $\mathcal{S}$.

2.6.1 Metrics

Supervised *ML* models, *AutoML* (Definition 9) or *NAS* (Definition 10) approaches, presented in the previous sections aim at optimising the predictive performance based on a true label, a model f and a given metric $\mathcal{L}$. Therefore, this (predefined) metric guides the optimisation process and is crucial to obtain suited results related to the metric. Furthermore, it indicates the performance of the model based on given data instances and thus enable the monitoring of models regardless if they were trained in an offline or online learning scenario.

> ➢ **Definition 16.** Supervised Learning Metric, adopted from Kulis [142]
> Given a set Y where $y \in Y$ refers to a label and $\hat{y} \in Y$ refers to the predicted label (i.ex of a model f), then a metric is a function $\mathcal{L} : Y \times Y \to \mathbb{R}$, where $\mathbb{R}$ is a set of real numbers, and for all $y, \hat{y}, z \in Y$ the following conditions are satisfied:
>
> 1. $\mathcal{L}(y, \hat{y}) \geq 0$ (non-negativity)
> 2. $\mathcal{L}(y, \hat{y}) = 0$, if and only if $y = \hat{y}$ (coincidence axiom)
> 3. $\mathcal{L}(y, \hat{y}) = \mathcal{L}(\hat{y}, y)$ (symmetry)
> 4. $\mathcal{L}(y, z) \geq \mathcal{L}(y, \hat{y}) + \mathcal{L}(\hat{y}, z)$.

In Definition 16, we define a supervised learning metric that assumes a label y; however, denote that a metric, in general, can be seen as a distance measure between two data points and thus is not necessarily bound to the supervised learning setting. It can be used on the one hand to train and optimise a *ML* system or, on the other hand, to evaluate and monitor the system's performance. In the following, we depict several commonly applied metrics for the classification, the regression task and for measurements going beyond the predictive performance. We first confine the introduction of the metrics to an already given data set $\mathcal{D}$ and finally show the adaptability to data streams.

2.6.1.1 Classification

Within the classification task, we consider that the label y from a data point $(\overrightarrow{x}, y)$ is categorical and thus has k finite many degrees of maturity. For the sake of simplicity, we will initially deal with a binary classification case with $k = 2$. The metrics can simply be extended to a multi-class classification metric by treating each class separately as a binary classification problem. Accordingly to Russell et al. [202] the performance of a model can be estimated by:

***TP* (*True Positive*):** Items $(\overrightarrow{x}, y)$ where the true label y is *positive* and the predicted class $\hat{y}$ is correctly classified as *positive*.

***FP* (*False Positive*):** Items where the true label y is *negative* and the predicted class $\hat{y}$ is wrongly classified as *positive*.

***TN* (*True Negative*):** Items where y is *negative* and the predicted class $\hat{y}$ is correctly classified as *negative*.

***FN* (*False Negative*):** Items where y is *positive* and the predicted class $\hat{y}$ is wrongly classified as *positive*.

Considering a multi-class classification as a set of many binary classification problems, the defined metrics can be estimated separately for each class by y and $\hat{y}$. Further, metrics based on the *TP*, *FP*, *TN* and *FN* considering multi-class classification can be estimated on a (i) macro-level or (ii) micro-level [202, 170]. Within the macro-level each class is given a uniform distributed weight to estimate a combined metric (e.g. F-Score) from all classes. On the micro-level all items within $\mathcal{D}$ are weighted equally and thus classes with more observations will have more influence on the combined metric. Following the optimisation problems defined within the definitions for supervised offline learning, *AutoML* and *NAS* that minimise a loss based on a validation protocol, we invert the search direction by multiplying each metric by -1 to comply with the optimisation problems. We consider in the following a micro-level averaging to define common classification metrics as follows:

Precision The precision metric measures the proportion of positive identifications that were correctly predicted over the number of *FP* plus the number of *FP* and is defined as follows:

$$\mathcal{L}^{(\text{precision})} = -\frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{k} TP_i + FP_i} \tag{2.23}$$

Recall The recall score is also known as sensitivity and measures the proportion of relevant instances that were correctly classified. In other words it measures the effectiveness of the model and is defined by:

$$\mathcal{L}^{(\text{recall})} = -\frac{\sum_{i=1}^{k} TP_i}{\sum_{i=1}^{k} TP_i + FN_i} \tag{2.24}$$

F-Score The F_β metric combines the *Recall* and *Precision* metric with a given parameter β. It measures the effectiveness of a model by weighting the *Recall* score β times as much importance as the *Precision* score. It can in dependence of β be defined as follows:

$$\mathcal{L}^{(F_\beta)} = -(1+\beta^2)\frac{\mathcal{L}^{(\text{precision})} * \mathcal{L}^{(\text{recall})}}{(\beta^2 * \mathcal{L}^{(\text{precision})}) + \mathcal{L}^{(\text{recall})}} \tag{2.25}$$

Commonly the harmonic mean (F_1) is applied, where the importance of *Recall* and *Precision* are equally distributed.

Accuracy The accuracy metric measures how close a model predicts $\hat{y}$ values to the labelled data point y and can be defined as follows:

$$\mathcal{L}^{(\text{accuracy})} = -\frac{\sum_{i=1}^{k} \frac{TP_i+TN_i}{TP_i+TN_i+FN_i+TN_i}}{k} \tag{2.26}$$

Comparing the *Accuracy* metric with the *Precision* metric, the *Accuracy* measures the closeness to a given target (correctness), whereas *Precision* measures the closeness to a set of labels (exactness).

Besides the presented metrics, other metrics, such as the *ROC* (*Receiver Operating Characteristic*) and its area under the curve, are popular measurements for binary classification to visualise the trade-offs between sensitivity (*Recall*) and specificity (*FP*-rate).

2.6.1.2 Regression

The regression task considers that the label y from a labelled data point $(\overrightarrow{x}, y) \in \mathcal{D}$ or $\in \mathcal{S}$ is continuous. Thus it is necessary to measure the performance of a model based on some distance measure. Contrary to the classification task in Section 2.6.1.1, the error measures in regression are already directed toward the introduced optimisation problems. For the sake of simplicity, we limit the regression metrics to single output values, where a single value represents the label y. Commonly used metrics in regression tasks are *MAE* (*Mean Absolute Error*), *MSE* (*Mean Squared Error*), R^2 and *MAPE* (*Mean Average Percentage Error*) [94, 170].

MAE The *MAE* calculates as the name states the mean absolute error between the true label y and the predicted label $\hat{y}$ and can thus be measured as follows:

$$\mathcal{L}^{(\text{MAE})} = \frac{1}{t}\sum_{i=1}^{t}|y_i - \hat{y}_i| \tag{2.27}$$

MSE The *MSE* estimates the models performance by a quadratic error estimation. This has the effect that stronger deviations of the prediction are more weighty in the error measure. Gradient-based methods in particular can thus better incorporate stronger deviations into the training.

$$\mathcal{L}^{(\text{MAE})} = \frac{1}{t}\sum_{i=1}^{t}(y_i - \hat{y}_i)^2 \tag{2.28}$$

MAPE One disadvantage of both, *MAE* and *MSE*, is that they are not normalised. However, the *MAPE* is defined as the percentage mean of the absolute difference between predicted values and true values divided by the true value y:

$$\mathcal{L}^{(\text{MAPE})} = \frac{1}{t}\sum_{i=1}^{t}\frac{|y_i - \hat{y}_i|}{max(\epsilon, |y_i|)} \tag{2.29}$$

In Equation 2.29, the variable $\epsilon << 1$ ensures that the division by zero is prevented. The application of the *MAPE* metric, however, has some significant drawbacks. If predictions $\hat{y}$ are too high in comparison to the ground truth y, the *MAPE* metric can exceed 1 and thus has theoretically no upper bound. Further, the *MAPE* metric penalises negative errors heavier when considering considering that y and $\hat{y} > 0$.

Coefficient of variance The R^2 metric computes the coefficient of determination and is calculated as follows:

$$\mathcal{L}^{(R^2)} = \frac{\sum_{i=1}^{t}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{t}(y_i - \mu_y)^2} - 1 \tag{2.30}$$

In Equation 2.30, μ_y denotes the mean of all $y \in \mathcal{D}$ or $\in \mathcal{S}$. It indicates how well unseen samples are likely to be predicted by the model and has a lower bound of 0 but can possibly be greater than 1. The best value is achieved with a R^2 score of 0.

2.6.1.3 Further Metrics

We defined performance metrics based on the previous section's classification and regression task. The input of the metric $\mathcal{L}(\cdot,\cdot)$ in both cases was a set of ground truth y and predicted $\hat{y}$ labels. When it comes to the application and deployment of *ML* algorithms, additional metrics become necessary that go beyond the characteristics of the differences between y and $\hat{y}$. For this purpose, instead of defining the input of the loss by the true label y and the prediction $\hat{y}$, the input can be extended to (i) the true label y, (ii) the corresponding features $\overrightarrow{x}$ and (iii) the model f. The loss metric is then defined by $\mathcal{L}(y, \overrightarrow{x}, f)$ and enables to measure performances that go beyond the pure estimation of the error based on the difference between y and $\hat{y}$. Exemplary, the following metrics can thus be modelled:

Latency By passing the features $\overrightarrow{x}$ and a model f to the metric, it is possible to measure time relevant metrics i.ex. the time the model needs to compute a prediction $\hat{y}$ usually measured in [ms] or [s] that is then used for training.

Training time Considering an *online learning* scenario, the model's training time can be measured by updating the model within the metric function by considering the inputs y and $\overrightarrow{x}$ as a training sample. The time the model requires to update can again be measured within the function.

Memory Usually, the models are instantiated within classes. Based on these classes, the memory consumption can be extracted within the metric. By adding the model's latency or the training time for an instance, the deployed RAM hours for a model f can be calculated and used as a metric.

In this section, we discussed several classification, regression or metrics that go beyond the error estimation based on a ground truth label y and the predicted label $\hat{y}$. The ability to apply the metrics shown in an *online learning* environment can be achieved through a similar approach to the *CMA* in Equation 2.21 [23]. The

question about the presented metrics' application remains open and is discussed in the following within the evaluation protocols.

2.6.2 Batch Evaluation

In terms of measuring a model's performance by a validation protocol $\mathcal{V}(f, \mathcal{L}, \mathcal{D}_{train}, \mathcal{D}_{test})$, it is essential to decide which part of the data is used for learning and which examples should be used to evaluate the model's performance. Knowing in advance which data is available offers some advantages when training and testing a model, such as enabling the flexible design of training and test data sets. To evaluate the generalisation of a model in an *offline learning* scenario, the model is first trained and then evaluated, whereby the data used for the evaluation was not considered within the training process. For this purpose, different methods of splitting the data into a training data set $\mathcal{D}_{train}$ and a validation set $\mathcal{D}_{valid}$.

The nearest option is the *holdout* evaluation, where the data set $\mathcal{D}$ is split into a training and a validation data set. The model is first trained on $\mathcal{D}_{train}$ and then evaluated based on a held-out data set $\mathcal{D}_{valid}$. *NAS* or *AutoML* would be performed by training different model configurations on the same data set $\mathcal{D}_{train}$ and evaluate them on identical validation splits. Usually the data set is split into 80% $\mathcal{D}_{train}$ and 20% $\mathcal{D}_{valid}$. However, this method has the drawback that the measurements $\mathcal{L}$ based on D_{valid} tend to be statistically non-representative and thus, tuning hyperparameters on fixed training and validation splits tend to overfit on $\mathcal{D}_{valid}$.

To overcome this problem, the k-*fold* cross-validation protocol shuffles the data set $\mathcal{D}$ and splits it into k subsets of equal size. By shuffling $\mathcal{D}$, it is assumed that the data set is *IID* (*Independent and Identically Distributed*). For each subset of the k-folds, we train a model f on the remaining $k-1$ splits and evaluate the performance based on the k^{th} split withheld for evaluation. The score of all k results is averaged to obtain a final metric score [202].

In the case where $\mathcal{D}_{train}$ is relatively small, *Iterated* k-*fold* cross-validation repeats the k- fold cross-validation protocol to further shuffle the subsets for each fold and averages again the scores obtained by each run of the k-fold cross-validations. However, the k- fold cross-validation protocol already trains k different models, which is already expensive. The *Iterated* k-*fold* cross validation trains and evaluates based on i iterations $i \times k$ models, which is even more expensive to compute. In Sections 2.3.2 and 2.3.3, we already presented that the definitions for *CASH* (Definition 9) and *NAS* (Definition 10) are based on a k-fold cross validation protocol. However, this validation protocol does apply to an *online learning* environment since the entire data set $\mathcal{D}$ has to be available in k-fold cross-validation. In the following, we depict evaluation protocols that are compatible with data streams $\mathcal{S}$.

2.6.3 Online Evaluation

In the case of data streams, not all data is available at once. Furthermore, the instances are often correlated with each other, which is referred to as time series and makes the use of batch validation protocols that shuffle the data set inappropriate to measure a model's performance. Further, when *concept drifts* occur, more recent data are more relevant than data points that occurred at the beginning of the data stream or beforehand of the *concept drift*. Despite the assumption that the data within the stream $\mathcal{S}$ is *IID* and k-fold cross-validation could be applied by caching data points from the stream, this approach is expensive to evaluate and thus infringes the memory and time requirements for online learning [23].

As the data stream is possibly infinite, the continuous learning of a model and the development of a metric $\mathcal{L}$ over time is of great importance. Furthermore, to measure the performance when *concept drifts* occur, recent data points within $\mathcal{S}$ might be more important than data points at the beginning of the process. Considering the *online learning* capabilities of a model f and a metric $\mathcal{L}$, the following evaluation procedures have become established [23] in *online learning* environments:

Holdhout: Similar to the holdout method in an *offline learning* scenario, small holdout data sets are generated from the data stream that is exclusively used to evaluate a model trained on the data stream. Thus, the evaluation of a model is performed periodically when a certain number of instances are available within the newly generated and updated data set. Nevertheless, this approach can lead to non-representative results if important change events cannot be captured due to the holdout. Further, a *concept drift* can only be identified within the evaluation metric $\mathcal{L}$ when a new data set has been generated from the data stream and held out from training.

Interleaved Test-Then-Train: The interleaved test-then-train method aims to solve the problem of held-out data sets and is applied to use all the data available for training and to evaluate a model f on a data stream. Following the *online learning* system in Figure 2.14, whenever a new instance arrives, it is first used to incrementally evaluate the current model based on $\mathcal{L}$ and then the instance is used for training the model f. On the one hand, all labelled data points within the data stream are used for both learning and evaluation. Moreover, no data needs to be held out and thus memorised from the data stream to obtain a model's performance.

Prequential: Assuming an infinite data stream, the interleaved test-then-train evaluation protocol is becoming increasingly resistant to deviations and, thus, in measuring *concept drifts*. Dawid [61] introduced the prequential evaluation protocol, which is composed of the terms predictive and sequential. However, the prequential evaluation protocol extends the interleaved test-then-train approach. It introduces a window size (e.g. sliding window or a decay factor) to assign a higher relative significance to recent data points. The size of the window and the decay factor are configurable at the beginning of the data stream and control the sensitivity to changes.

Interleaved chunks: Following the interleaved test-then-train approach, interleaved chunks stores small data batches in the sequence of the data stream, whereby the sizes of the chunks to evaluate a model f can be of different sizes.

Referring to the *online learning* requirements, the general interleaved test-then-train, as well as the prequential evaluation protocol operates as follows given a metric $\mathcal{L}$, a model f and features $\overrightarrow{x}$ with their corresponding label y from a data stream $\mathcal{S}$ [23]:

1. Get a labelled instance $e_t = (\overrightarrow{x}_t, y_t)$ from the data stream $\mathcal{S}$.
2. Predict $\hat{y} = f(\overrightarrow{x})$.
3. Update $\mathcal{L}$ based on $\hat{y}$ and y or on f, $\overrightarrow{x}$ and y.
4. Train f incrementally based on the labelled data point $(\overrightarrow{x}, y)$
5. Proceed to the next instance e_{t+1}.

It becomes clear that different evaluation techniques exist in the streaming context, which depends on the application and the properties to be evaluated. This, however, must be determined individually and in dependence on the area of application.

In this section, we depicted two areas essential for this work. First, we introduced metrics that measure classification and regression model's predictive and further performances. These metrics aim to represent the utility of a model based on its predictive performance or metrics that go beyond the differences between y and $\hat{y}$. Further, these metrics aim to represent different features and characteristics of a model's performance. However, end-user's or domain expert's utilities are diverse in that they may follow combinations of metrics. Thus, the question of a user-based adaptation remains open. Second, we introduced different validation protocols for an offline and online learning scenario and highlighted their differences. However, these validation protocols emphasise the influence of the choice of validation protocol.

2.7 Summary

Summarising the foundations of this work, we started from a *DM* point of view by defining a *ML* pipeline in Section 2.2 and introducing commonly used pipeline components for the *offline learning* setting. Following the concatenation of these pipeline elements, we formalised and introduced the concepts behind *AutoML* as well as *NAS*. For the subsequent personalisation of *AutoML* and *NAS* approaches, we presented in Section 2.4 ranking approaches and switched in Section 2.5.1 from an *offline learning* setting to an *online learning* point of view. Within an *online learning* environment further requirements defined by Bifet et al. [23] emerged to apply *ML* algorithms on data streams. To finalise the foundations of this work, we defined in Section 2.6 metrics and validation protocols for both online and offline learning scenarios that enable the evaluation of models f as well as *ML* pipelines, *AutoML*, and *NAS* approaches. Building on the foundations, we depict the research relevant to this work in the following.

3

Related Work

In this chapter, we provide the research work that is related to (i) utility-centric and to (ii) online learning based *AutoML* system. In Section 3.1, we discuss *Metric Learning* as research field that is related to the utility-based adaptation. The field of *Metric Learning* incorporates research around *HGML* (*Human Guided Machine Learning*). A further related research field towards a utility-centric adaptation is *Multi-Objective ML*, where the goal is to learn a model based on multiple target functions. Section 3.2 provides the work for *Multi-Objective ML* systems. Both research fields are related in that they incorporate multiple metrics and thus implicitly enable the optimisation towards a certain utility. To present the related of an online *AutoML* system, we present in Section 3.3 techniques for ensemble learning. *Online ensemble* methods are related to *online AutoML* in that they consider various (homogeneous or heterogeneous) models to adapt to the underlying data distribution of the data stream. Further, in Section 3.4, we depict the related work that applies *NN*s by assuming an *online learning* scenario and show the diversity in its application. However, the related work towards the applicability of *NN*s in *online learning* further motivates the development of an *online DL* framework, that aims to unify and foster the research on *NN* within *online learning*.

3.1 Metric Learning

In this section, we investigate *metric learning*, but also *HGML* (*Human Guided Machine Learning*) that aims to guide *ML* models towards a certain utility that might be pursued. The research field of *metric learning* aims to learn a metric $\mathcal{L}^*$ that is especially in this research field, also referred to as distance metric or similarity measurement. Further, *metric learning*, as well as *HGML*, are related in that they aim to provide a metric $\mathcal{L}$ that performs better than the initially selected metric [142]. *Metric learning* aims to optimise this metric toward a specific task, such as the supervised offline learning task defined in Definition 1. While we aim to learn a metric that represents an end-users or domain experts utility of a *ML* model, *metric learning* is broadly explored in unsupervised learning tasks, such as clustering, where the aim of *metric learning* is to find a suitable distance measure that separates features $\overrightarrow{x}$ within a data set. Thus, approaches that adopt a metric can be distinguished by the task they aim to solve or the similarity they seek to measure. Further, common approaches that incorporate a *ML* model can be distinguished whether they learn a new underlying metric based on the given data set $\mathcal{D}$ or on (human) feedback. In Figure 3.1, we illustrate the common process for *metric learning* and refer to a (human) feedback U_{train}. This commonly human feedback is often referred to as *HGML* (*Human Guided Machine Learning*) [33]. It goes beyond pure *metric learning* approaches in that not only the metric but also the data, and thus the underlying model is adapted. Furthermore, *HGML* often incorporates a user interface that enables end-users or domain experts to interact with the underlying system.

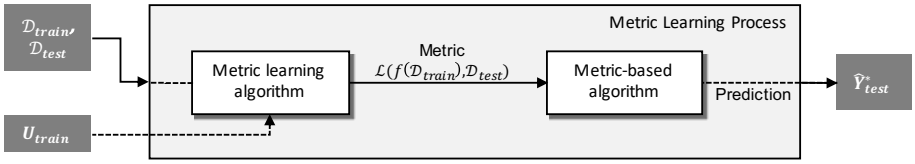


Figure 3.1: Common process for *metric learning*

Assuming feedback-based approaches the feedback can be (i) implicit, (ii) explicit or (iii) mixed feedback [33]. Implicit feedback influences the features $\overrightarrow{x}$ of an underlying data set $\mathcal{D}$ and thus the performance of the underlying model. For instance, for image classification tasks, some image parts might be more relevant than others and based on feedback that highlights these areas, e.g. by cropping the image towards the relevant parts of the image. Explicit feedback influences the metric by giving feedback based on the model's output $\hat{y}$. For example, an end-user or domain expert gives feedback on whether a model's output is correct or not. Mixed feedback methods combine both implicit and explicit feedback methods, e.g. by giving explicit feedback on a model's performance and implicit feedback based on whether the explicitly given feedback follows a specific pattern [33].

Furthermore, these approaches can be differentiated based on their evaluation. The performance of a *metric learning* approach can be measured based on subjective or objective measures. Objective performance evaluations evaluate "how well the [...] machine co-operations performs to achieve a task is to compare the *ML* system with its [non-modified] counterpart." [33, p. 346 f.]. Subjective metrics are based on questionnaires, surveys or interviews that contain subjective information about the suitability of a newly learned metric.

Table 3.1: Selected related work for *metric learning* (upper part) based on Bellet et al. [15] and Xing et al. [246] as well as for *HGML* (lower part) based on Boukhelifa et al. [33]. The *Task* column indicates which learning task is addressed by the related work and the *Model* indicates the underlying model that is used for evaluation. The column *Data Centric* indicates whether the metric learning approach is learned based on a given data set or on external feedback (*HGML*). The *Feedback* column states if the approach incorporates implicit and explicit feedback. Finally, the *Metric* relates to the utility (subjective or objective) used for evaluation purposes.

Approach	**Task**	**Model**	**Data Centric**	**Feedback**		**Metric**	
				impl.	expl.	subj.	obj.
Xing et al. [246]	clustering	*k-NN*	✓	×	×	×	✓
Goldberger et al. [97]	classification	*k-NN*	✓	×	×	×	✓
Qi et al. [191]	classification	*k-NN*	✓	×	×	×	✓
Weinberger and Saul [237]	classification	*k-NN*	✓	×	×	×	✓
Gil et al. [95]	-	-	✓	✓	×	×	×
Azuan et al. [7]	classification	-	×	✓	×	✓	×
Heimerl et al. [108]	classification	*SVM*	×	✓	×	✓	✓
Ehrenberg et al. [72]	classification	-	×	×	✓	✓	✓
Endert et al. [76]	clustering	-	×	✓	×	×	×
Bryan et al. [40]	classification	*PCA*	×	×	✓	✓	✓
Koyama et al. [136]	reduction	BFGS[1]	×	✓	✓	✓	✓
Dabek and Caban [60]	clustering	*k-NN*	×	×	✓	✓	✓
Boukhelifa et al. [34]	reduction	*EA*	×	✓	✓	✓	×

[1] Broyden–Fletcher–Goldfarb–Shanno optimisation

To depict the basic commonalities and differences for *metric learning* and *HGML*, we present in Table 3.1 selected work that builds the preliminary related work towards utility-based adaptation where the metric is modified. However, *metric learning* emerged in 2002, where Xing et al. [246] formulated *metric learning* as an optimisation problem and measures the difference between a ground truth label y and the prediction $\hat{y}$ by

$$\mathcal{L}(y, \hat{y}) = \mathcal{L}_M = ||y - \hat{y}||_M = \sqrt{(y - \hat{y})^T M (y - \hat{y})}. \tag{3.1}$$

The goal becomes to optimise based on a data set $\mathcal{D}$ a semi definite matrix M under the restriction that $\mathcal{L}^2(y, \hat{y}) \leq 1$ (see Definition 18). Since this data-centric modification includes the data set $\mathcal{D}$, it can be set as a hyperparameter or be integrated within the underlying model and thus implicitly learns the underlying metric. Further, these approaches are mainly based on *k-NN* models. *k-NN* models have the advantage that they store the data set and choose the k nearest neighbours based on the underlying metric $\mathcal{L}$. This makes it a suitable algorithm for searching for appropriate metrics for evaluation purposes. Goldberger et al. [97] propose a stochastic *k-NN* approach where the underlying metric is optimised based on the expected error of the *k-NN* model. Qi et al. [191] propose an efficient sparse metric learning approach that is specifically useful in the case of high-dimensional data and Weinberger and Saul [237] suggest a large margin nearest neighbours classifier that as well as a *SVM* classifier minimise the distance of data points that share the same label and maximises the margin between data points of different labels within $\mathcal{D}_{train}$.

HGML goes beyond pure data-centric *meta learning* approaches in that it aims to guide based on a learned metric (see Figure 3.1) a *ML* model. While the data-centric approaches are mainly evaluated based on *k-NN* methods, *HGML* is often assessed based on a gold standard metric (objective) or the impact on user (subjective) satisfaction. Thus, *HGML* goes one step further and evaluates the entire system depicted in Figure 3.1. This enables the evaluation of the metric's impact on the underlying *ML* model and the construction of utility-specific distance metrics. Further, the field of *HGML* compromises different goals, where Gil et al. [95] present the requirements for a *ML* system, where domain experts use their knowledge to affect how *ML* systems work. Azevedo and Santos [6] showed that implicit feedback inferred from user corrections could be as impactful as explicit feedback by modifying the underlying data set without applying *ML* models. For text classification, Heimerl et al. [108] propose an interactive approach in order to complement search and filter techniques and evaluate their approach based on *SVM*s. While Heimerl et al. and Azevedo and Santos integrate an implicit feedback, where the *ML* model's output is influenced by modified data, Ehrenberg et al. [72] present an explicit feedback system denoted as *DDLite*. This framework provides a platform for creating, evaluating, and debugging labelling functions based on predefined metrics and is assessed based on *k-NN* clustering. Within a user study, Endert et al. [76] propose a semantic interaction method to support analytical reasoning. However, this study does not incorporate an evaluation of the impact for *ML* models. Within *ISSE* Bryan et al. [40] incorporate human feedback to separate recorded speech from a cell phone based on a time-frequency visualisation and evaluate their approach by applying *PCA*. Based on images, Koyama et al. [136] propose an iterative and interactive approach to support colour enhancement based on images. By applying a non-linear optimisation technique (*BFGS*), the proposed system enhances images to a user's intention. Dabek and Caban [60] present a method for modelling user interactions within a *ML* model to automatically generate suggestions in a visualization system and Boukhelifa et al. [34] present a user guided *ES* approach that combines predefined metrics and user's input to visualise pertinent views to the user.

In this section, we presented the related work towards data-centric metric learning as well as *HGML* that mainly incorporates implicit or explicit human feedback. However, while data-centric metric learning approaches can be integrated as a hyperparameter configuration of the underlying model, *HGML* approaches that enable the adaption of *ML* systems towards a desired utility are evaluated on isolated algorithms and neglect the hyperparameter configuration.

3.2 Multi-Objective AutoML

A line of research strongly related to a utility-driven *AutoML* system that enables to take into account multiple objectives within the search process is *multi-objective AutoML* Instead of learning a metric that guides the *ML* system in the direction of a particular utility or stating a preference based on human feedback *HGML*, the idea of multi-objective *AutoML* and *NAS* is to optimise a set of metrics and thus to fulfil a utility by optimising a Pareto frontier. The search goal in multi-objective *NAS* is, therefore, to approximate the Pareto-frontier, which expresses the degree to which available objectives can be jointly met. We define the multi-objective *AutoML* problem as multi-objective *CASH* problem as follows:

➢ **Definition 17.** Multi-Objective *CASH* Problem
Let $\mathbf{L} = \{\mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(l)}\}$ be a set of available and complementary metric functions. Further, let

$$\overrightarrow{A}^*, \overrightarrow{\lambda}^* \in \underset{\overrightarrow{A} \in \mathcal{A}^{|g|}, \overrightarrow{\lambda} \in \Lambda}{\arg\min} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(\mathcal{P}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}(\mathcal{D}_{\text{train}}^{(i)}), \mathcal{D}_{\text{valid}}^{(i)}) \tag{3.2}$$

be the single-objective *CASH* problem defined in Definition 9 with a set of step independent algorithms $\mathcal{A} = \{A^{(1)}, \ldots, A^{(R)}\}$, their parametrisation domain Λ and a structure g that defines a *ML* pipeline as $\mathcal{P}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}$, then the goal of multi-objective *AutoML* is to optimise the single-objective *CASH* problem for all $\mathcal{L} \in \mathbf{L}$. Further, a *ML* pipeline $\mathcal{P}^{(1)}$ Pareto-dominates another pipeline $\mathcal{P}^{(2)}$ if the following applies to $\forall j \in 1, \ldots l$:

$$\frac{1}{K} \sum_{i=1}^{K} \mathcal{L}^{(j)}(\mathcal{P}^{(1)}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}(\mathcal{D}_{\text{train}}^{(i)}), \mathcal{D}_{\text{valid}}^{(i)}) \leq \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}^{(j)}(\mathcal{P}^{(2)}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}(\mathcal{D}_{\text{train}}^{(i)}), \mathcal{D}_{\text{valid}}^{(i)}) \tag{3.3}$$

Pareto-optimal pipeline configurations are those configurations that are not Pareto-dominated by any other configuration. The set of all Pareto optimal configurations is denoted as Pareto-frontier [73].

With the definition of the multi-objective *CASH* problem, we can simply extend this definition to a multi-objective *NAS* problem, where a *ML* pipeline $mathcal P_{g, \overrightarrow{A}, \overrightarrow{\lambda}}$ is exchanged with a neural architecture $f_{g, \overrightarrow{z}, \overrightarrow{\lambda}}$ defined in Definition 10. In Table 3.2, we present the related work for *multi-objective AutoML* and *multi-objective NAS* approaches. While *AutoML* systems are carried out on commonly available resources without *GPU* or *TPU* acceleration, *NAS* and especially *multi-objective NAS* are computational very expensive. This need for accurate and computational cheap neural architectures has led to intensified research in the field *multi-objective NAS*.

Table 3.2: Selected related work for *Multi-Objective AutoML* (upper part) and *Multi-Objective NAS* (lower part). The *Solver* column indicates the underlying solver or algorithm used to search within the search space (*Structure*). *Ensemble* indicates, whether the approach uses ensembles for prediction and *Parallel* indicates whether the algorithm is executed in a parallel manner. The *Benchmark* column indicates if the approach uses or introduces a Benchmark and *Time* of the approach is sensitive to the runtime of the system during training.

Approach	Solver	Structure	Ensemble	Parallel	Benchmark	Time
Olson et al. [177]	*GP*	Variable	✓	✓	×	✓
Pfisterer et al. [187]	*BO*	*XGBoost*	✓	×	×	×
Gardner et al. [93]	*GP*	-	✓	×	×	×
Olson et al. [177]	*GP*	MLP	✓	✓	×	✓
Hsu et al. [114]	*RL*	*AlexNet*	×	×	×	∼
Elsken et al. [73]	*EA*	Variable	×	✓	✓	×
Kim et al. [128]	*EA*	Variable	×	✓	×	✓
Chu et al. [53]	*EA*	Variable	✓	×	×	∼
Mendoza et al. [167]	*BO*	*FNN*	×	×	×	∼
Lu et al. [159]	*GP*	VGG	×	×	✓	∼
Dong et al. [66]	*BO*	Variable	×	×	×	∼

To configure *ML* pipelines based on multiple metrics, Olson et al. [177] extend *TPOT* in that the *GP* creates a set of Pareto-optimal *ML* pipelines that is retrieved from the populations of each generation. However, *TPOT* is capable of building, based on the created Pareto-frontier, an ensemble of pipelines and searches the configurations space in a parallel manner. Since it is a framework that is implemented to find suitable *ML* pipelines for unseen data sets, it does not incorporate benchmark data sets. The time sensitivity while training implicitly includes the complexity of the *ML* pipeline created. To optimise the *ML* pipelines towards their complexity, e.g. their length or the size of the ensemble, *TPOT* proposes to integrate a time-sensitive metric. Another approach for *multi-objective AutoML* is proposed by Pfisterer et al. [187] which uses *BO* extended by sub-evaluations to include multiple metrics and thus to build a Pareto-frontier. This approach is build on *Autoxgboost* [227], whereby it automatically configures a *XGBoost* instance and thus do not incorporate a *ML* pipeline as defined in Definition 3. However, Pfisterer et al. [187] do not incorporate a time sensitive evaluation for their proposed approach. Finally, Gardner et al. [93] propose a multi-objective and constrained approach that builds on a *EA* approach. It creates a Pareto-frontier based on different *TP* and *FP* rates and thus does not consider time restrictions. As Pfisterer et al., Gardner et al. do not consider a *ML* pipeline but a search space based on various different models.

Especially for real-world applications, multi-objective *NAS* is a central tool for balancing available constraints, such as sufficiently high predictive accuracy, low energy consumption, or fast execution. This led to the development of various works towards *multi-objective NAS* that incorporate the training time in the form of *FLOPS* (*FLoating point Operations Per Second*). As *TPOT* [177] supports *NAS* in form of the automatic configuration of *MLP* networks, it is listed in Table 3.2 twice. In *NAS* it uses the same underlying *GP* algorithm as for *AutoML*. However, it differs from other *NAS* approaches in that it searches *MLP* network architectures of variable size and thus is commonly used on tabular data sets used for classification and regression tasks. A large part of the related work relates to image classification and consider data sets such as *CIFAR*-10, *CIFAR*-100 or *ImageNet*. Existing approaches aim at efficiently maximizing sets of objectives $\mathbf{L}$ by exploiting *RL* [114], *EA* [73, 128, 53, 159] or *BO* [167, 66]. Within *MONAS*, Hsu et al. [114] use a *RL* agent that propose based on an encoding different neural architectures. It varies the structure based on *AlexNet* [137] different neural architectures. The reward for the *RL* agent is given by a linear combination of

prediction accuracy and other metrics that incorporate *FLOPS*. Elsken et al. [73] propose within *LEMONADE* an approach that searches a Pareto-frontier by applying lamarchian *EA*. Further, it introduces a Benchmark search space for further researches in *multi-objective NAS*. However, *LEMONADE* does not consider a time or resource-sensitive evaluation but utilises a warm-stating mechanism incorporating weight sharing. *Nemo*, proposed by Kim et al. [128], optimise simultaneously time and accuracy for *CNN* architectures and Chu et al. [53] propose within *MONAS* a *multi-objective NAS* approach that applies *NSGA-II* on a cell-based search space and incorporate *FLOPS* as well as the number of trainable parameters of the neural architecture. Mendoza et al. [167] use fixed *FNN* parameter configuration space and evaluate their approach based on the training time and error rate on tabular data sets. Further, the approach uses a *SMAC* as the underlying optimiser. *NSGA-Net*, proposed by Lu et al. [159] propose as Chu et al. [53] *NSGA-II* as underlying solver to handle trade-off among multi-objectives. Further, it uses *BO* to profit from the search history. Finally, Dong et al. [66] propose in *PPPNet* an *EA* approach that incorporates a *RNN* to regress a pursued multi-objective metric and thus to select in each generation suitable neural architectures.

Comparing the proposed *multi-objective AutoML* and *NAS* approaches the drawback of Pareto-optimization comes apparent. Considering large Pareto-frontiers, that already in single-objective *NAS* approaches are computationally expensive, comparisons as stated in Definition 17 become even more expensive. A further drawback is that it assumes that the end-user is fully aware of all objectives and is capable of interpreting the Pareto-front.

3.3 Online Ensemble Learning

In this section, we present the related work towards *online ensemble* learners. Ensemble techniques are related to *AutoML* in that they use, accordingly to Definition 6 a set of models to learn a joint function $\hat{r}$ that performs better than an individual model. However, these models can either be single models or *ML* pipelines without the goal of optimising their configuration. While Definition 6 is applicable on *online ensemble algorithms*, not all ensemble techniques defined in 2.2.4.6 are applicable on data streams. In Table 3.2, we provide an overview of the related work towards ensemble learning techniques. We divide the proposed ensemble techniques into the underlying model used within the evaluation and the strategy to build the ensemble. Further, we investigate whether the ensemble technique uses homogeneous or heterogeneous underlying models and whether they use a sliding window or a change detector to adapt to data streams. While *Voting* [133, 21, 26, 169] , *Bagging* [181] and *Boosting* are commonly applied ensemble techniques, *Stacking*, where different models are aggregated by training new models based on the predictions made, is not applied within *online learning* environments. However, in [180], Oza and Russell provide an experimental comparison towards online and batch-based bagging and boosting methods and show the superiority of *online ensembles*. The nature of data streams even opens new ensemble techniques such as proposed by van Rijn et al. [233].

Table 3.3: Related work for *online ensemble learning* techniques, considering the underlying (i) Model for evaluation, the ensemble (ii) Strategy, whether the ensemble is capable of using (iii) Heterogeneous Models and if the ensemble applies (iv) Sliding Windows or (v) Change Detectors.

Approach	Model	Strategy	Heterogen Models	Sliding Window	Change Detector
Oza and Russell [181]	*MLP & NB*	bagging & boosting	×	×	×
Kolter and Maloof [133]	*NB*	voting	×	×	×
Bifet et al. [26]	*HT*	voting	×	✓	✓
Bifet et al. [24]	*HNBT*[2]	voting	×	✓	✓
Minku and Yao [169]	Decision Tree	voting	✓	✓	✓
Brzezinski and Stefanowski [41]	*HT*	-	×	✓	✓
van Rijn et al. [233]	-	last best	✓	✓	×
Gomes et al. [98]	*Random Forest*	voting	×	✓	✓
Gomes et al. [99]	*HT*	voting	×	✓	✓
Bahri et al. [9]	*k-NN*	-	×	✓	✓

Oza and Russell [181] propose *OB* (*Online Bagging*) and *Online Boosting* as ensemble techniques on data streams. *OB* updates a set of models by weighting each instance from the stream with a $Poisson(1)$ distributed number. In order to illustrate the similarities and differences between our approach presented in Section 9.3 and *OB*, we depict in Algorithm 1 the *OB* algorithm.

Algorithm 1 Online Bagging, adopted from Oza and Russell[181]

Input:
Data stream S, Set of models P,
Output:
Prediction: $\hat{y}$

if e_t **then** ▷ Start Data Stream
  for $f \in P$ **do** ▷ Update each model
    $k \leftarrow Poisson(1)$
    loop ▷ Repeat k times
      f.fit(e_t)
    end loop
  end for
end if

The *Online Boosting* algorithm proposed by Oza and Russell [181] simulates sampling with replacement from the *batch learning* approach. Further, it updates the weights for each model's error and estimates the prediction $\hat{y}$ based on these weights. Kolter and Maloof [133] propose a dynamic weighted majority voting that creates and removes base algorithms in response to changes in performance. This dynamic weighted majority voting is evaluated on *NB* classifiers but supports similar to *online bagging* and *boosting*, heterogeneous models, without using sliding windows or change detectors. In "New Ensemble Methods for Evolving Data Streams" [26], Bifet et al. propose to use a *bagging* ensemble of *HT* with different tree

[2] Hoeffding Naive Bayes Tree

sizes. Further, *LB* (*Leveraging Bagging*) is proposed in [26] and refined in [24], where the sampling k (see Algorithm 1) follows a $Poisson(6)$ distribution and an *ADWIN* change detector is applied to remove the worst model of the ensemble and to replace it with a new model. Minku and Yao [169] propose with *DDD* an ensemble method that operates in two modes: prior and posterior drift detection. Instead of an *ADWIN* change detector an *EDDM* is applied. The modes of operation are switched, whether the *EDDM* detector detect a *concept drift* or not. The approach is evaluated based on 25 decision trees. A generic block ensemble is proposed by Brzezinski and Stefanowski [41], where the data stream is partitioned into blocks to train each model within the ensemble accordingly to a models quality measure. An ensemble method that builds on *OB* and uses heterogeneous ensembles is *BLAST* (Best Last) proposed by van Rijn et al. [233]. *BLAST* selects from an *ensemble* of models an active model that is used for prediction. However, the active model is determined by the past performances of each model of the ensemble. *ARF* (*Adaptive Random Forest*), proposed by Gomes et al. [98] use an ensemble of *Random Forests* models that is trained and adapted regarding an *ADWIN* change detector and a warning detector. Further, Gomes et al. propose in "Streaming Random Patches for Evolving Data Stream Classification" [99] *Streaming Random Patches (SRP)* that combines random subspaces and bagging while using a strategy to detect drifts similar to the one introduced in *ARF* [98]. Bahri et al. [9] adapt *k-NN* models to data streams, whereby the disadvantage of storing the data is reduced by random projection

Concluding the related work for *online ensembles*, most approaches employ homogeneous algorithms with identical configurations. Further, they do not consider pipeline configurations. Assuming algorithm and hyperparameter search spaces, such as employed in *autosklearn* ($|\Lambda| = 110$ possible configurations) in an *offline learning* environment, training all possible base algorithms within ensembles (e.g., *OB* [181] and *LB* [24]) becomes increasingly inefficient. Further, they do not consider search heuristics to determine a combination of suitable algorithms within the algorithm search space. However, to consider larger search spaces, the related work provides some concepts that lead to *online AutoML*.

3.4 Online Deep Learning

In Section 2.5.4.3, we already depicted the general applicability of *NN*s as part of the Foundations. This section, however, presents the related work for approaches that apply *NN*s in an *online learning* environment and illustrates the broad range of applications for *NN* in *online learning*. Further, the related work for *online learning NN*s illustrates the heterogeneity of the applications that we aim to homogenise within the *online DL* framework. However, we base this part of the related work on the survey papers Jain et al. [123] and Pérez-Sánchez et al. [186] and limit to *NN* for *supervised learning* scenarios. Jain et al. [123] presents the related work published between 2003 and 2013 and highlights the used architecture types as well as the application domains. Pérez-Sánchez et al. [186] provide a review of techniques that adapt to concept drifts and states the main strategies to face concept drifts are (i) sliding windows, (ii) instance weighting and (iii) ensembles. This review includes concepts and strategies toward structural adaption (*NAS*) methods. However, empirical evaluations are pending, and the suitability of *NAS* to face concept drifts is not presented. In Table 3.4, we provide an overview of selected approaches that are related to the application of *NN*s in an *online learning* environment. For a complete overview we refer to the survey provided by Jain et al. [123] and Pérez-Sánchez et al. [186].

Table 3.4: Selected related work for *online DL* techniques, considering the (i) Application, the (ii) Architecture, the underlying (iii) Optimiser and whether the evaluation of the approach is sensitive to (iv) concept drifts.

Approach	Application	Architecture	Optimiser	Change
Lee et al. [148]	Evacuation	*FNN*	-	✓
Sheng Wan and Banta [210]	Medicine	*MLP*	*PIL*	×
Akhbardeh et al. [3]	Medicine	*ART*	-	×
Zhou and Lai [251]	Finance	*MLP*	*SGD*	✓
Ergen and Kozat [77]	Finance	*LSTM*	*SGD*	×
Lobo et al. [157]	-	*SNN*	-	✓
Sahoo et al. [204]	-	*MLP*	*HBP*	×
Baydin et al. [14]	-	*MLP*	*SGD*, *Adam*	×
Nose et al. [175]	Mobility	*CNN*	*SGD*	×
Lobo et al. [156]	-	*SNN*	-	×
Liu et al. [154]	-	*RBF*	*SGD*	✓

Considering the related work for *NN*s in an online learning environment, recent approaches either focus on new neural architecture types [153, 3] or the application of new optimizers for specific neural architectures [148, 236, 157]. Both are crucial for the application in real-world scenarios. As data streams often have the characteristics of time series where the instances of the data stream are not stationary but partially dependent on their order of arrival. For instance, this requires a neural architecture that can account for time dependencies. Further, when *concept drifts* occur, a fast strategy to adapt to the changing pattern is necessary. This, however, compromises the development of optimisers that adapt the weights of a neural architecture when *concept drifts* occur. Lee et al. [148] propose a General Regression Neural Network (*GRFNN*) that is applicable on data streams. This approach was developed to predict the evacuation time of a karaoke centre in the event of a fire and employed incremental learning to reduce its computational requirements. Sheng Wan and Banta [210] propose a *PIL* (*Parameter Incremental Learning*) approach where the strategy is that in the process of adapting the network to training data by adjusting its parameters, the past learning steps of the networks are also explicitly required to be perceived to a certain extent. In comparison to *SGD*, *PIL* shows in [210] a faster convergence of a *MLP* architecture to the data stream. Another architecture used in *online learning* is the *ART* (*Adaptive Resonance Theory*) network, where a central feature is a pattern matching process that compares an external input with the internal memory of a network. Akhbardeh et al. [3] use *ART* networks to classify cardiac cycles. However, *ART* networks are commonly used in health care [123] in an unsupervised learning scenario. Zhou and Lai [251] present a model based on *EMD* that incrementally learns and forecasts gold markets. *EMD* is used to divide a time series into different subsets and then to perform a back-propagation step on a *MLP* network. Ergen and Kozat [77] evaluate *LSTM* architectures and predict stock prices as well as the exchange rates in an incremental manner. Besides *SGD*, Ergen and Kozat applies particle swarm optimisation techniques in order to incrementally update the weights of the *LSTM* architecture. Referring to the functioning of the human brain and the premise that we are exposed to constant streams of data, Lobo et al. [157] propose a *SNN* architecture that evolves over the data stream. *SNN*s incorporate the concept of time in that they do not transmit information at each propagation step of the data stream. In this approach, the *SNN* architecture and the weights of the architecture evolve based on a sliding window and a *ADWIN* drift detector. Lobo et al. evaluate their approach based on various real-world data streams that range from electricity to weather forecasts. In "Spiking Neural Networks and Online Learning: An Overview and Perspectives", Lobo et al. [156] further present an overview of applications of *SNN* architectures in an *online learning* environment. In this review, the ability to adapt to the *concept drifts* is addressed.

Recent approaches address the adaptation of the *NN* underlying optimiser to data streams [204, 14]. Sahoo et al. [204] further addresses the *vanishing gradient* problem in *DL* networks with multiple layers and proposes an approach that dynamically evolve the depth of the *FNN*. The *HBP* (*Hedge Backpropagation*) optimiser optimises the weights as well as the smoothing parameters of Softmax functions inserted in each layer of the network. In "Online Learning Rate Adaptation with Hypergradient Descent", Baydin et al. [14] extend *SGD* and *Adam* to a hyper-gradient learning rate that reduces the time and resources needed to tune the learning rate by simultaneously showing faster convergence. This optimiser is applied to *MLP* networks in an online learning scenario, as well as on *CNN* architectures in an *offline learning* setting. Another approach that applies *CNN* but in an *online learning* setting is proposed by Nose et al. Nose et al. [175]. The goal is to train a lane-keeping assistant incrementally, where the input is a sequence of images annotated by a driver's steering input. While Baydin et al. takes the resource consumption into account and thus considers the requirements defined by Bifet et al. [23], Nose et al. gives a structural approach without considering the resource consumption. Finally, Liu et al. [154] propose a *NN* architecture based on *RBF* (*Radial Bias Function*) that automatically replaces the worst performing hidden nodes within the *NN* and compares this approach against *LSTM* architectures. It considers changes within the data stream based on a synthetic Rossler chaotic time-series data stream but does not incorporate *drift detection* algorithms such as *ADWIN*.

In order to enable *NAS* in an *online learning* environment, Liu et al. and [157] propose first systems, that dynamically adapt a specific network type to a data stream. However, considering multiple network types or a framework that enables the simple application of *NAS* on different data sources as depicted in Section 2.3.3, further research needs to be conducted. The related work, presented in Table 3.4, also shows the different areas of applications, whereby each application is an isolated solution and does unfortunately not consider a generic framework to open research across the application, architecture and the underlying optimiser.

3.5 Summary

This chapter presented the related work towards a utility- and stream-based adaptation of *ML* and *AutoML* systems. For a utility-based adaptation, we presented in Section 3.1 *Metric Learning* and *Human Guided Machine Learning* as concepts that enable the adaptation of *ML* systems or pipelines towards a certain utility by implicitly or explicitly learning a metric function $\mathcal{L}^*$ Further, in Section 3.2, we presented *Multi-objective ML* approaches that are related in that they aim to optimise a set of objectives by retrieving a Pareto-frontier and thus reflecting the end-user's or domain expert's utility. Searching a Pareto-frontier, however, might remain computationally expensive. As stated in Definition 17, the larger the Pareto-front becomes, the more solutions need to be considered and evaluated regarding all metrics within $\mathbf{L}$ in each iteration for possible improvements. Mainly when the pursued utility comprises many metrics, the advantage of metric learning approaches becomes clear.

Towards stream adaptation, Section 3.3 presents the related work towards online ensemble learning methods. This part of the related work shows that recent developments in ensemble learning for *online learning* scenarios showed impressive results in terms of performance and adaptivity to *concept drifts*. However, only a few approaches consider heterogeneous base models to build an ensemble, and none considers *online learning ML* pipelines. Finally, in Section 3.4, we presented, following the foundations for *online NN* depicted in Section 2.5.4, the related work towards *online DL* models. A common framework is required to consolidate the research that has been conducted and make research in this area perspectively comparable. The presentation of the related work in this chapter supports the research questions presented at the beginning of this thesis in Section 1.3 and shows the significance of the contributions toward utility- and stream-based adaptation of *ML* and *AutoML* systems incorporating *DL* methods.

Part III

Utility Adaptation

"All models are wrong, but some are useful."

–Box [35], 1979

4

Preference Learning

This chapter investigates preliminarily Hypothesis I and is about learning preferences that combine different already available metrics and thus about the adaption to a particular utility. It mainly builds on the publications "Personalized Automated Machine Learning" (ECAI - 2020) and "Personalized Neural Architecture Search" (ICDMW - 2021), where the general approach towards utility adaption in both publications is similar but both investigate different systems (*AutoML* and *NAS*) and require thus different metrics and evaluations.

> **Hypothesis I** (Utility Adaptation)
> *Existing approaches for* AutoML *and* NAS *aim efficiently maximising individual or sets of objectives $\mathcal{L}$. By variation of the target function $\mathcal{L}$, the output of* AutoML *systems can be adapted and tailored to the needs of the user and thus to a utility.*

With a retrospective of the provided foundations and the related work, this chapter aims to provide an approach that is able to vary the underlying target function of a *AutoML* or *NAS* system. Within the foundations, we systematically defined a *ML* pipeline and showed the vast configuration space of *AutoML* but also *NAS* systems. Further, we presented in Section 2.6 different metrics that might be pursued and showed the heterogeneity of these metrics that, in the first place, measure the distance between two data points. To include resource-sensitive goals, we stated in Section 2.6 an extension to the formalisation of a metric $\mathcal{L}(\cdot, \cdot)$ that incorporates not only two instances ($\overrightarrow{x}$ or y), but also the model f ($\mathcal{L}(\cdot, \cdot)$) that can with regard to the aim of this thesis be a neural architecture or a *ML* pipeline $\mathcal{P}$. In Chapter 3, we depicted the related work and presented different approaches that incorporate different metrics within the learning process, namely (i) *metric learning*, (ii) *HGML* and (iii) *multi-objective ML*. The related work showed that *metric learning* is commonly applied on *ML* models that incorporate a distance metric (e.g. *k-NN*), *HGML* is mainly applied on user interaction tasks and *multi-objective AutoML* and *NAS* frameworks aim at taking a small number of metrics into account. Thus, the goal is to depict an system that might answer, how an *AutoML* (RQ I) or *NAS* (RQ II) system can be adapted to a utility an end-user or domain expert might pursue. In order to depict a utility-based adaptation, we assume in this chapter again an *offline learning* scenario, where all data is available at the beginning of the learning process. Assuming an online learning scenario for preference elicitation, the underlying utility might evolve over time as a data stream. This further increases the complexity and blurs the evaluation.

First, we formalise in Section 4.1 the problem following the *CASH* problem (see Definition 9) and the *NAS* formalisation (see Definition 10) defined within the Foundations in Section 2. We introduce the general approach in Section 4.2 by its components for learning new metrics $\mathcal{L}^*$ based on a pairwise ranking model and introduce its components within the Sections 4.2.1 - 4.2.4. In the following chapters of this part, we then present an *AutoML* and *NAS* sensitive evaluation.

4.1 Problem Formalisation

In this section we formalise the problem towards a utility-based adaptation of *AutoML* and *NAS*. While the goal of *HGML* is to create systems that allow domain experts to use domain knowledge to steer the search for a suitable model, the aim of *metric learning* is to learn a metric that measures the distance (or performance) of data points and thus the model's performance. Derived from both research fields, *HGML* and *metric learning*, we can set the following requirements, which were already roughly depicted at the beginning of this work.

Requirements II. Utility-based *ML*, derived from Gil et al. [95] and Li and Tian [150] :

R II–1. Certain variables and parameters of the underlying model may be given more priority by an end-user.

R II–2. The end-users utility should reflect within the metric.

R II–3. A utility-based metric should influence the optimisation of an underlying model in the direction of the utility

R II–4. The utility score should follow a symmetry.

R II–5. The utility score should be non-negative.

The first requirement (Requirement *R II–1*) describes the ability of an end-user to be able to state their preference towards their priority [95] (or utility). This requirement thus concerns the access to a system that can be adapted to the user's needs. Requirement *R II–2* is about the metric that should reflect the utility. To measure the fulfilment of this requirement, we can measure the performance regarding the distance of a predefined utility and the predicted utility by the newly generated metric. In order to steer an *AutoML*, *NAS* or *ML* system, the newly generated metric should not only reflect the end-user's or domain expert's utility, but also it should have an impact on the optimisation process of the underlying optimisation process (Requirement *R II–3*). The Requirements *R II–4* and *R II–5* are derived from Li and Tian [150] and are general requirements to a metric. However, these metrics are often neglected in practice, but compliance with these has advantages in certain circumstances. A symmetric metric (Requirement *R II–4*) where the order of the metric's inputs has no impact on the metric's score ($\mathcal{L}(y_i, y_j) = \mathcal{L}(y_j, y_i)$) has the advantage, when considering a metric $\mathcal{L}(\cdot, \cdot)$ and one input y as ground truth and one input $\hat{y}$, that differences in prediction and ground truth are symmetric and also evaluated symmetrically. Considering a metric as distance measurement, Requirement *R II–5* becomes apparent. To avoid questioning whether a score of 1 is equal, better or worse than a score of -1, the advantages of a non-negative metric become apparent.

With regard to the Requirements II, the advantages and disadvantages of different approaches towards utility adaptation presented in this work (*HGML*, *metric learning* and *multi-objective ML*), we base our utility-based *AutoML* and *NAS* system on a *metric learning* approach. In Definition 18, we formally define our *metric learning* approach that learns a metric based on a set of heterogeneous metrics.

➢ **Definition 18.** Metric Learning
Let $\mathbf{L} = \{\mathcal{L}^{(1)}, \dots, \mathcal{L}^{(l)}\}$ be a set of available metric functions, then the goal becomes to learn a novel (metric) function:

$$\mathcal{L}^* : \mathcal{L}^{(1)} \times \cdots \times \mathcal{L}^{(l)} \rightarrow \mathbb{R}^+ \tag{4.1}$$

By assuming that one can model a function $\phi : \mathbf{L} \rightarrow \mathbb{R}^l$ which generates a feature vector for the available set of metrics, we can reduce learning a novel metric $\mathcal{L}^*$ to a regression problem, where we attempt to learn

$$h_{\mathcal{L}^*} : \phi(\mathbf{L}) \rightarrow \mathbb{R}^+ \tag{4.2}$$

As in Definition 17 for the *multi-objective CASH* problem, our *metric learning* approach receives a set of available metric functions $\mathbf{L}$. The goal is to learn a new metric function $\mathcal{L}^*$ where $\hat{h}_{\mathcal{L}^*}$ is an approximated regression model for the novel metric function $\mathcal{L}^*$. We have to learn weights $\theta \in \mathbb{R}^{|\mathbf{L}|}$, e.g. $\hat{h}_{\mathcal{L}^*} = \phi(\mathbf{L})^T\theta$ for the linear case. This learning task can further be integrated within the *CASH* as well as the *NAS* problem. In Definition 19, we define the *utility-based CASH* problem based on the *metric learning* definition in Definition 18, we aim to solve within this chapter.

➢ **Definition 19.** Utility-based *CASH* and *NAS* problem.
Combining the *CASH* problem from Definition 9 and the metric learning problem from Definition 18, then the resulting utility-based *CASH* problem can be defined as follows.

$$g^*, \overrightarrow{A}^*, \overrightarrow{\lambda}^*, \mathcal{L}^*, \in \underset{\overrightarrow{A} \in \mathcal{A}^{|g|}, \overrightarrow{\lambda} \in \Lambda, \mathcal{L} \in \mathbf{L}}{\arg\min} \frac{1}{K} \sum_{i=1}^{K} \hat{h}_{\mathcal{L}^*}(\mathcal{P}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}(\mathcal{D}_{\text{train}}^{(i)}), \mathcal{D}_{\text{valid}}^{(i)}) \tag{4.3}$$

Extending the *NAS* problem (see Definition 10) where a loss function $\mathcal{L}$ is assumed to be given with the metric learning problem from Definition 18, then the resulting utility-based *NAS* problem can be defined as follows.

$$g^*, \overrightarrow{z}^*, \overrightarrow{\lambda}^*, \mathcal{L}^* \in \underset{g \in G, \overrightarrow{z} \in \mathcal{Z}, \mathcal{L} \in \mathbf{L}}{\arg\min} \frac{1}{K} \sum_{i=1}^{K} \hat{h}_{\mathcal{L}^*}(f_{g, \overrightarrow{z}, \overrightarrow{\lambda}}(\mathcal{D}_{\text{train}}^{(i)}), \mathcal{D}_{\text{valid}}^{(i)}) \tag{4.4}$$

where $\hat{g}_{\mathcal{L}^*}$ is the approximated regressor for the novel metric function $\mathcal{L}^*$ for which we have to learn weights $\theta \in \mathbb{R}^l$, e.g. $\hat{h}_{\mathcal{L}^*} = \phi(\mathbf{L})^T\theta$ for the linear case.

In Definition 19 the search for an optimised metric $\mathcal{L}^*$ is integrated within the search for an optimal parametrisation of *AutoML* ($g^*, \overrightarrow{A}^*$ and $\overrightarrow{\lambda}^*$) in Equation 4.3 and of *NAS* (g^*, $\overrightarrow{z}^*$ and $\overrightarrow{\lambda}^*$) in Equation 4.4. However, the integration of the *metric learning* problem into the *CASH* or *NAS* system does not necessarily mean a simultaneous optimisation of all parameters. In the following, we provide a system that aims to solve the Problems defined within this section.

4.2 Approach

In this section, we depict the overall system towards the utility-based adaptation of *AutoML* and *NAS* that is designed to consider (human) feedback. We consider a *metric learning* approach and inject a pairwise ranking model as metric function $\mathcal{L}^*$ into the *AutoML* and *NAS* system as shown in Figure 4.1.

First, we present the interactions between the system's components, as well as the overall algorithm. Then we depict the components of the systems accordingly to the in and outputs of the components defined in Algorithm 2 in more detail.

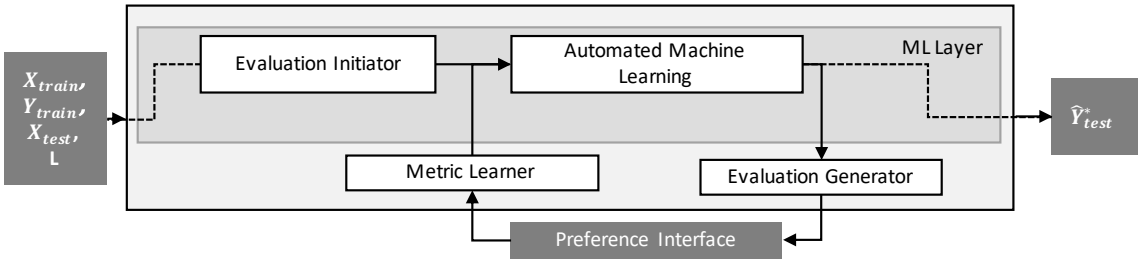


Figure 4.1: Approach towards an utility-based adaptation of *AutoML*, incorporating *NAS*.

The system depicted in Figure 4.1 consists of five components and takes as stated in the foundations in Section 2.3.2 a data set $\mathcal{D}$ as well as a set of metrics $\mathbf{L}$. Furthermore, accordingly to the process for *metric learning* systems, depicted in Figure 3.1, our approach incorporates (human) feedback from a *Preference Interface*. The main components of the system are an *Evaluation Initiator*, a single objective *AutoML* system, an *Evaluation Generator*, a *Preference Interface* and a *Metric Learner*. The *Evaluation Initiator* component generates a diverse set of *ML* pipelines or *NAS* systems by calling *AutoML* instances regarding different metrics within $\mathbf{L}$. The *AutoML* component is referred to a single-objective *AutoML* system, that searches for suited *ML* pipelines or neural architectures following one predefined metric $\mathcal{L}$. *ML* pipelines or neural architectures created and trained by the *AutoML* (incorporating *NAS*) system are further processed by the *Evaluation Generator*. This component processes the *ML* pipelines in that it evaluates the pipelines regarding all metrics $\mathcal{L} \in \mathbf{L}$ or generates user interpretable visualisations. We refer to these visualisations that contain all information an end-user or domain expert requires to state their preference to as Segments U (see Figure 3.1). Within the *Preference Interface*, an end-user or domain expert states their preference by rating the proposed Segments in a pointwise, pairwise or list-wise manner. Rated Segments are referred to as U^{judged} and processed within the *Metric Learner* component. The *Metric Learner* learns accordingly to Definition 18 a new metric $\mathcal{L}^*$. This utility-based metric is then passed in a last step to the *AutoML* system, which then optimises the underlying *ML* pipeline or neural architecture towards the utility of the end-user or domain expert stated within the *Preference Interface*. To learn an underlying metric, we base our approach on pairwise comparisons. This approach is, in comparison to the other proposed ranking approaches in Section 2.4 preferable, as within pointwise preference elicitation approaches, an end-user or domain expert is assumed to be able to state the metric score of a pipeline configuration without knowledge about the performances of other pipeline configurations. Further, a list-wise approach, where an end-user or domain expert, states his or her preference based on ordered sublists, can be reduced to multiple pairwise comparisons.

Algorithm 2 Utility-based *AutoML*

1: **Input:**
2: Dataset $\mathcal{D}_{\text{train}} = \{X_{\text{train}}, y_{\text{train}}\}$,
3: Dataset $\mathcal{D}_{\text{valid}} = \{X_{\text{valid}}, y_{\text{valid}}\}$,
4: Set of Metrics $\mathbf{L}$,
5: Number of pairwise comparisons ω
6:
7: **Output:**
8: Fitted *ML* pipeline $\mathcal{P}^*$
9:
10: \\ Generate set of pipeline configurations P :
11: $P \leftarrow \emptyset$ ▷ Initialise set of pipeline configurations
12: $S \leftarrow \emptyset$ ▷ Initialise set of segments
13: **for** $\mathcal{L} \in \mathbf{L}$ **do**
14: $f \leftarrow$ *AutoML*$(\mathcal{L}, X_{\text{train}}, y_{\text{train}})$ ▷ Fit *AutoML* instance
15: $P_{\mathcal{L}} \leftarrow$ *AutoML*$.get\_pipelines()$ ▷ Get all evaluated $\mathcal{P}_{g,\overrightarrow{A},\overrightarrow{\lambda}}$
16: $P \leftarrow P \cup P_{\mathcal{L}}$
17: **end for**
18: \\ Generate segment pairs U^{pair} :
19: $U^{pair} \leftarrow SegmentGenerator(P, \mathcal{D}_{valid}, \omega)$ ▷ see Algorithm 4
20: \\ Get users preference U^{judged} :
21: $U^{judged} \leftarrow UserPreference(U^{pair})$ ▷ see Section 4.2.3
22: \\Train learning to rank, see Section 4.2.4
23: $X^{(1)}, X^{(-1)} SampleComparisons(U^{judged})$
24: $\mathcal{L}^* \leftarrow RankNet.fit(X^{(1)}, X^{(-1)})$
25: $f^* \leftarrow$ *AutoML*$(\mathcal{L}^*, X_{\text{train}}, y_{\text{train}})$ ▷ Train *AutoML* instance on $\mathcal{L}^*$
26: **Return** $\mathcal{P}^*$

In Algorithm 2, we depict the overall process towards an utility-based adaptation of *AutoML*. This adaptation can be extended to *NAS* by replacing a pipeline configuration $\mathcal{P}_{g,\overrightarrow{A},\overrightarrow{\lambda}}$ with a neural architecture $f_{g,\overrightarrow{z},\overrightarrow{\lambda}}$. However, within Chapter 6, we propose in Algorithm 5 an integrated *NAS* optimisation process based on *ES* that is sensitive to the number of generated neural architectures in that it uses an initial population of neural architectures for the generation of pairwise segments U^{pair} and for the evolutionary process. As for the *multi-objective* case of *AutoML*, Algorithm 2 takes as *AutoML* systems a data set $\mathcal{D}$, split into $\mathcal{D}_{train}$ and $\mathcal{D}_{valid}$. Furthermore, it takes a set of metric $\mathbf{L}$ that of metrics and a predefined value ω for the number of pairwise comparisons. The output of the algorithm is a to a utility suited *ML* pipeline $\mathcal{P}^*$. The different phases of Algorithm 2 are depicted within the components in the following sections.

4.2.1 Evaluation Initiator

Within the initialisation phase (Algorithm 2, l.11ff.) the *Evaluation Initiator* generates different pipeline configurations $\mathcal{P}_{g,\overrightarrow{A},\overrightarrow{\lambda}}$. In order to obtain a diverse set of pipeline configurations for each metric $\mathcal{L}^{(i)} \in \mathbf{L}$ an *AutoML* instance is trained on $\mathcal{D}_{train}$. We denote the pipeline configurations obtained by an *AutoML* instance based on a specific metric as $P_{\mathcal{L}^{(i)}}$ and the set of all generated configurations as P. Denote that within the initialisation, we obtain not only the best pipeline configuration $\mathcal{P}^*_{\mathcal{L}^{(i)}}$ of an *AutoML* instance fitted on $\mathcal{L}^{(i)}$, we obtain all evaluated pipeline configurations evaluated during training. Thus the number of pipeline configurations obtained by the *AutoML* system is significantly higher than the number of metric functions $|\mathbf{L}|$. The aim of the *Evaluation Initiator* is thus to generate a set of pipeline configurations that is as diverse as possible for the later preference elicitation approach. We now describe the *Evaluation Generator*,

which is called after the initialisation phase (Algorithm 2, l.19), and that takes the generated set of pipeline candidates P, a validation data set, and the number of segments to generate.

4.2.2 Evaluation Generator

The aim of the *Evaluation Generator* component is to process the generated pipeline configurations and to obtain Segments that can be visualised within the preference interface component. Thus, this component takes the generated pipeline configurations P and the number of segments ω that should be generated. We denote a segment s_i as tuple $(X_{valid}, y_{valid}, \mathcal{P}^{(i)}_{g,\overrightarrow{A},\overrightarrow{\lambda}})$ that contains the data set $\mathcal{D}_{valid}$ in form of the features X_{valid} and the ground truth labels y_{valid} as well as pipeline configuration $\mathcal{P}^{(i)}_{g,\overrightarrow{A},\overrightarrow{\lambda}}$. A segment enables to obtain all performance measurements or metrics an end-user or domain expert might pursue. For example, performance metrics such as the accuracy can be obtained by performing $\mathcal{L}^{(accuracy)}(y_{valid}, \mathcal{P}^{(i)}_{g,\overrightarrow{A},\overrightarrow{\lambda}}(X_{train}))$ (see Equation 2.26). As we pursue a pairwise preference elicitation approach, where an end-user or domain expert states their preference by pairwise comparisons (see. Section 2.4), the question of which segments to compare for ranking also takes part. In Algorithm 3, we depict the process for generating segment pairs U^{pair}. It executes a random generation of segment pairs. However, to train a *metric learning* algorithm with preferences, it might be favourable to choose within the pairwise setting segments that are ranked relatively close to each other. Techniques that train an algorithm based on the uncertainty of the algorithm are called *active learning* techniques. These could further boost the performance of the *metric learner* in order to converge faster to the underlying utility. However, to avoid losing the scope of this work and putting the general framework and function into perspective, we base ourselves on a naive, random approach. In Algorithm 2, the *Evaluation Generator* is executed in line 9.

Algorithm 3 Segment Generation

1: **Input:**
2: Set of configurations P,
3: Number of segment pairs to be generated ω,
4: Validation dataset $D_{\text{valid}} = \{X_{\text{valid}}, y_{\text{valid}}\}$,
5:
6: **Output:**
7: Set of segment pairs U^{pair}
8:
9: **while** $|U^{\text{pair}}| \leq \omega$ **do**
10: $i, j \leftarrow$ Select random $i, j \in P$
11: $s_i \leftarrow (X_{\text{valid}}, y_{\text{valid}}, \mathcal{P}^{(i)}_{g,\overrightarrow{A},\overrightarrow{\lambda}})$
12: $s_j \leftarrow (X_{\text{valid}}, y_{\text{valid}}, \mathcal{P}^{(j)}_{g,\overrightarrow{A},\overrightarrow{\lambda}})$
13: **if** $(s_i, s_j) \notin U^{pair}$ **then**
14: $U^{\text{pair}} = U^{\text{pair}} \cup \{(s_i, s_j)\}$
15: **end if**
16: **end while**
17: **Return** U^{pair}

In Algorithm 3, we first select two random pipelines (or architectures) configurations from a set of configurations. For both configurations, a segment s is generated and combined to a tuple (s_i, s_j). A segment contains all the information necessary to make a preference selectable for the end-user or domain expert. If the tuple (s_i, s_j) is not already in the set of segment pairs U^{pair}, the tuple is appended to the set U^{pair}. This process is repeated until the number ω of necessary segment pairs U^{pair} is reached. Denote that the

maximal number of segment pairs is given by all possible combinations of $P^{(i)} \in P$, without taking into account the order of segments within the segment pairs U^{pair}.

4.2.3 Preference Interface

Considering the generated segment pairs U^{pair} from the *Segment Generation* component, the *Preference Interface* component is the interface between the end-user and the domain expert. Thus, this component visualises the generated segment pairs U^{pair} so that the user is capable of judging which segment in U^{pair} is preferred to the other one in the pairwise setting. The tuple $(X_{valid}, y_{valid}, \mathcal{P}^{(i)})$ of each segment is visualised in that metrics, labels, or other performances are generated based on the configuration $\mathcal{P}^{(i)}$. In Figure 4.2, we provide an exemplary visualisation of segment pairs, where the end-user or domain expert could decide whether to choose the left or right segment and thus the preferred pipeline or architecture configuration.

Figure 4.2: Visualisation Preference Interface for *AutoML* based on the OpenML 179 [124] data set, where the task is to classify whether a person's yearly income is over 50K or not.

The exemplary *Preference Interface* in Figure 4.2 is based on the OpenML 179 data set. The task in this data set is to classify whether a person's income is over 50K a year or not. The visualisation contains a confusion matrix as well as a tabular visualisation of the ground truth label y_{valid}, the predicted label $\hat{y}$ from the configuration. Further, it contains on which features X_{valid} these predictions have been executed. The interface enables the end-user or domain expert to state their preferred configuration based on the provided information or to calculate on the already judged segment pairs U^{judged} a new metric. Besides the computation of performance metrics, such as the accuracy, the *Preference Interface* component can compute latency metrics or in *NAS* architecture-specific features, which enables a user-centric comparison of pipeline or architecture configurations. Performing *NAS* on image data sets, further, enables i.ex. the integration of explanations of the underlying configuration. Such a visualisation that explains why a configuration works the way it does is exemplarily provided by Grad-Cam [208]. In Figure. 4.3, Grad-Cam [208] visualises regions that a *CNN* architecture configuration payed attention on. Grad-Cam estimates important regions of an image by considering activation layers of a *NN*'s configuration before the output layer. Based on the visualisation of the performance metrics, as well as the activation functions, the end-users or domain experts can state their preference within pairwise comparisons as depicted in Figure 4.2. However, suggesting the domain expert or end-user suitable visualisations is a separate field of research we explicitly excluded from this work. Instead, we provide through the notation of a segment $S_i = (X_{valid}, y_{valid}, \mathcal{P}^{(i)}_{g\vec{A},\vec{\lambda}})$ the possibility to generate all kinds of visualisations an end-user might find helpful and thus offer a clearly defined system boundary. As depicted in Algorithm 2, the *Preference Interface* retrieves judged segment pairs, denoted to as U^{judged}, where all judgments from an end-user are stored as triples (s_i, s_j, c). The variable c corresponds to the end-users or domain experts preference, where $c \in \{-1, 1\}$. When the left segment was chosen, c equals -1, and when the right segment was selected, c equals 1. Considering the Requirements II, the *Metric Learner* component and the segments within U^{pair} enable Requirement *R II–1*, in that an end-user is able to give pipeline or neural architecture configurations based on their parameters and variables more priority.

Figure 4.3: Illustration for feature importance on *NASNet* based on Grad-Cam.

Given the judged segment pairs U^{judged}, the *Metric Learner* component is called to learn based on the judgements, a new utility-based metric $\mathcal{L}^*$

4.2.4 Metric Learner

Within the *Metric Learner* component a new metric $\mathcal{L}^*$ is trained based on the judged segment pairs U^{pair}. Thus, this component is the core component towards a utility-based *AutoML* and *NAS* system. It gets a set of judgments U^{judged} as input to generate a new, utility-based metric function $\mathcal{L}^*$. Since we assume a pairwise metric learning approach, we chose *RankNet* [44] as the underlying ranking approach (see Section 2.4.2), where a siamese *NN* takes the segment pairs as input to maximise the span between the judged segment pairs. Further each segment $s \in U^{judged}$ contains a pipeline or architecture configuration $P^{(i)} \in P$ and thus we can compute for all configurations $P^{(i)} \in P$ all predefined metric scores $\mathcal{L} \in \mathbf{L}$. The metric scores are then used for the pairwise *LTR* (*Learning To Rank*) approach. To use an already trained *LTR* model within this component as a utility-based metric function $\mathcal{L}$, it generates, based on the calculated metric scores, a

new set of features x. In Algorithm 4, we depict the scoring function $\mathcal{L}^*$, where a set of predefined metrics are used to generate the features for the underlying *RankNet* metric function.

Algorithm 4 Rank based Scoring function $\mathcal{L}^*$

1: **Input:**
2: Ground truth y, and prediction $\hat{y}$
3: Trained RankNet model $\text{NN}_{\text{RankNet}}$
4: Set of metric functions $L = \{\mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(l)}\}$
5:
6: **Output:**
7: Metric $\mathcal{L}^*(x)$
8:
9: Initialize $X \leftarrow \emptyset$
10: Initialize $\mathcal{L}^* \leftarrow \text{NN}_{\text{RankNet}}$
11: **for** $\mathcal{L}^{(i)} \in \mathbf{L}$ **do**
12: $\quad x \leftarrow x \cup \mathcal{L}^{(i)}(\hat{y}, y)$
13: **end for**
14: **Return** $\mathcal{L}^*(x)$ \\prediction

However, to train the underlying *RankNet LTR* approach (see Algorithm 2, l.21), two data sets $X^{(1)}$ and $X^{(-1)}$ needs to be generated. Similar to an already trained scoring function in Algorithm 4, the data sets are generated based on the pipeline configuration of each segment s in U^{judged}. Thus, both data sets $X^{(1)}$ as set of the preferred configuration and $X^{(-1)}$ as set of the corresponding non-preferred configurations have the same shape.

Following a *supervised offline learning* setting, the *RankNet* algorithm is trained on training data sets $X^{(1)}_{train}$ and $X^{(-1)}_{train}$. The structure of *RankNet* is divided into a siamese base structure and a meta-network. The base network proposed by Burges et al. [44] is a *CNN* architecture evaluated on *NLP* tasks. As in our case, we aim to rank pipeline or neural architecture configurations by a set of metrics $\mathbf{L}$; we chose a *MLP* architecture with one hidden layer and a *ReLU* activation layer to comply with the non-negativity requirement of metrics. The output of the base network takes a data set X. It is used in Algorithm 4 to score based on $\mathbf{L}$, $\mathcal{D}_{valid}$ and the pipeline or neural architecture configurations the performance of the configuration. The meta-network connects two siamese base networks that share their weights simultaneously. However, on base network scores on $X^{(1)}_{train}$ and the other on $X^{(1)}_{train}$. The difference between both scores distinguishes if the predicted scores provides a correct ranking within $X^{(1)}_{\text{train}}$ and $X^{(-1)}_{\text{train}}$ or not. While training, the aim of *RankNet* is to maximise within the meta-network the difference between the scores of $X^{(1)}_{train}$ and $X^{(-1)}_{train}$ achieved within the base networks that share their weights. Considering e.g. a judgement $(s_1, s_2, -1)$ we calculate for s_1 $L_1 = \{\mathcal{L}^{(1)}_1, \ldots, \mathcal{L}^{(l)}_1\}$ and for s_2 all metrics $L_2 = \{\mathcal{L}^{(1)}_2, \ldots, \mathcal{L}^{(l)}_2\}$ [139]. From the *Preference Interface* we know, that s_1 is ranked lower than s_2. To generate the training data set the features of s_2, generated on basis of $\mathbf{L}$ are added to $X^{(1)}_{\text{train}}$ and the generated features of s_1 are added to $X^{(-1)}_{\text{train}}$. The resulting data sets from all judgements U^{judged} of the *Preference Interface* are then used to train a *RankNet* instance. Since both base networks share their weight, it is irrelevant which base network is later used as a scoring function of $\mathcal{L}^*$. Regarding the last layer of the base network, we full-fill the

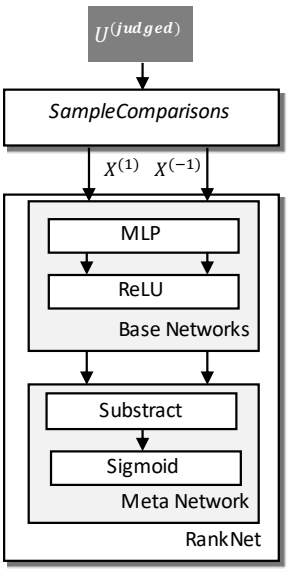


Figure 4.4: Integration of RankNet [44] within the Metric Learner.

metric Requirement *R II–5*, that $\mathcal{L}^*$ cannot become negative. Furthermore, the *RankNet* model follows by definition and due to its siamese architecture a symmetry in that $\mathcal{L}^*(X^{(1)}, X^{(-1)}) = \mathcal{L}^*(X^{(-1)}, X^{(1)})$. In Algorithm 4, we defined following the metrics presented in Section 2.6.1 the input as $\mathcal{L}^*(\cdot, \cdot)$. This input can, as also discussed in Section 2.6.1, be extended in that the function takes three (X,y and the configuration $\mathcal{P}$) inputs.

The generated metric function $\mathcal{L}$ that is trained towards an end-user or domain expert utility is in Algorithm 2 finally used to train an *AutoML* or *NAS* instance towards the utility.

4.3 Summary

In this section, we presented our approach toward a utility-based adaptation of an *AutoML* or *NAS* system. To incorporate a diverse set of metrics, we base this system on a *metric learning* approach, where we integrate a pairwise *LTR* system, namely *RankNet*, into our system. Around a *single-objective AutoML* or *NAS* system, we depicted the *Evaluation Initiator*, that generates based on a predefined set of metrics $\mathbf{L}$ a diverse set of configurations. These pipeline or neural architecture configurations are preprocessed for the *Preference Interface* within the *Evaluation Generator* component. Within Section 4.2.3, we illustrated the broad possibilities of the *Preference Interface* but also excluded the graphical preparation of segments from this work. The *Metric Learner* component takes the preferences from the *Preference Interface* and trains a *RankNet LTR* algorithm. The *RankNet* model learns a utility-based metric and is finally used to execute a *AutoML* instance towards the utility-based metric $\mathcal{L}^*$. Regarding the Requirements for *utility-based ML* an end-user is able to give certain variables or parameters of a *AutoML* or *NAS* system more priority by indicating his preference in a pairwise manner. Further, the *Metric Learner* component follows a symmetry (Requirement *R II–4*) and is non-negative (Requirement *R II–5*) owing to the *RankNet* architecture. However, the requirements, whether a utility is reflected by the *RankNet* approach (Requirement *R II–1*), as well as the impact of a utility-based metric on the underlying *AutoML* or *NAS* system (Requirement *R II–3*) are part of the following evaluation.

5

Automated Machine Learning

This chapter provides an empirical evaluation of our approach toward a utility-based adaptation of *AutoML*. Considering the requirements for utility-based *ML*, the Requirements *R II–1*, *R II–4* and *R II–5* are already fulfilled by the systems architecture. To answer RQ I, we provide in Section 5.1 a brief recap of the research questions and in Section 5.2 the experimental setup depicting the data sets and predefined metrics used for evaluation. In Section 5.3, we evaluate whether the *Metric Learner* component is capable to learn a certain metric (RQ I.1) as well as if this newly generated metric is steers an *AutoML* instance into the direction of the metric (RQ I.2).

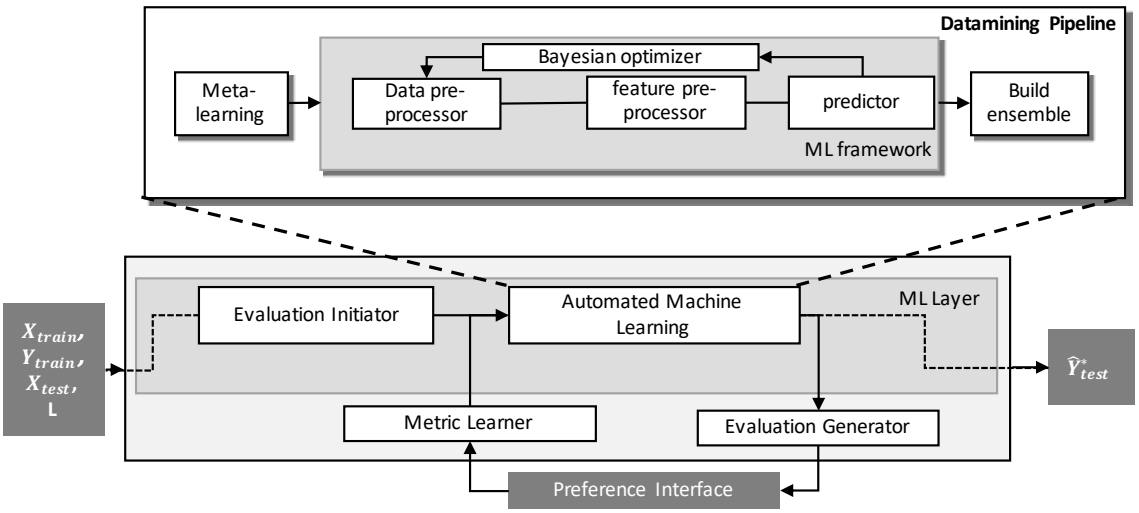


Figure 5.1: Approach towards a utility-based *AutoML* System

In Figure 5.1, we depict the overall system incorporating an *AutoML* system such as *autosklearn*, *TPOT* or *h2o*. The *AutoML* system takes besides the training data set $\mathcal{D}_{train}$ and a metric $\mathcal{L}$ that steers the *ML* pipeline creation process into the metrics direction. The results of this chapter have been published in "Personalized Automated Machine Learning" [139].

5.1 Recap Research Questions

Within the foundations provided in Chapter 2 it has become clear that *AutoML* systems are diverse in their functionality, but their potential is enormous. They generate *ML* pipelines considering data preprocessing, feature engineering as well as model selection. Further, they consider pipelines of variable length and thus go beyond pure *HPO*. However, the search for suitable *ML* pipelines needs a search direction to find pipelines

that fit the requirements and expectations an end-user or domain expert might pursue. In particular, we investigate in this chapter the following main research question through the experiment:

RQ I How can an *AutoML* system be adapted to a utility an end-used might pursue?

Regarding the requirements towards a utility-based adaptation of *ML* derived from Gil et al. [95] and Li and Tian [150], RQ I is two-folded and can be split into the research questions:

RQ I.1 How can a new target function $\mathcal{L}^*$ be learned?

RQ I.2 How can we optimise an *AutoML* system for metrics beyond established metrics?

The first research question, RQ I.1, is elementary for a utility-based *AutoML*. In Chapter 4, the overall system was already depicted. Within this system the *Metric Learner* component (see Figure 5.1) takes a key role to steer an *AutoML* system into the direction an end-user or domain expert might pursue. Within the *Metric Learner* component, proposed in Section 4.2.4, we already referred to a *LTR* approach that learns a ranking of *ML* pipelines created from an *AutoML* instance. However, pursuing RQ I.1 will shed light on whether the proposed *RankNet LTR* approach is able to learn a new target function $\mathcal{L}^*$. Further, this research question can be answered in several directions. One direction is, whether the underlying *LTR* approach is able to learn linearly or even more complex connections between predefined metrics. Another direction concerns the parameter ω in Algorithm 2 and the question about how many pairwise comparisons are necessary to learn these connection and thus the underlying utility. An empirical evaluation towards the *Metric Learner* component is presented in Section 5.3.1.

RQ I.2 concerns the overall system and aims to investigate the influence of different learned metrics on the performance of the system. This includes the influence of a learned metric on a diverse set of predefined metrics and investigates the sensitivity of *AutoML* systems towards the underlying metric. If, for example, the choice of the metric has only a minor influence on the *ML* pipelines performance, the additional components of our approach, including the *Metric Learner* component, would be redundant. This could be the case when the search space of the *AutoML* system is too small or does not bring any changes in performance when changing the configuration. To measure the impact of a learned metric on the optimisation process of the *AutoML* system, we provide in Section 5.3.2 an empirical evaluation of the overall system depicted in Figure 5.1. The overarching research question (RQ I) can finally be answered by combining both evaluations and thus showing the capability of our approach in adapting to a specific utility.

5.2 Experimental Setup

This section aims to provide the experimental setup for our approach. As depicted in Figure 5.1, our approach takes, besides a data set $\mathcal{D}$, a set of metrics or preferences that an end-user might pursue. In Section 5.2.1, we present the data sets and in Section 5.2.2 we define the set of metrics $\mathbf{L}$ on which we evaluate our approach. We executed all experiments on an *Intel(R) Xeon(R) Platinum 8180M CPU* with 2.50 GHz base clock and 1.5 terabytes of RAM without consideration of further *GPU* acceleration.

5.2.1 Data Sets

We base the evaluation of our approach on five different data sets retrieved from the *OpenML* [234]. The data sets are chosen based on the evaluation data sets from the *autosklearn* [81, 80] and *TPOT* [177] frameworks.

Further, we assume in all data sets a classification task, where the set of labels y is categorical and finite. All data sets are accessible based on the *OpenML* API[1]. The data sets range from real-world healthcare (thyroid disease) to artificially generated and simulated tasks.

Thyroid Disease: The thyroid disease data set was first introduced by Quinlan et al. [194]. It has 9172 data points containing 15 categorical and six continuous attributes. The features range from age, sex, and sickness to medical treatments. The task is to determine whether a patient is hypothyroid referred to the clinic. However, the data set contains three classes referring to a healthy functioning thyroid gland, a subnormal functioning and a hyper-function. Further, the data set is corrupted because it contains 5.4% missing values. As in *OpenML*, we refer to this data set by its id 38.

Quake: The Quake data set (*OpenML* id 772) is a binary classification data set that contains 2178 instances. The data set is provided by the National Earthquake Information Center, which determines the location and the size of all significant earthquakes worldwide. Thus, it has three features, where the task is based on the longitude, latitude and focal depth to determine whether the eruption source is an earthquake or another source. Since earthquakes are proportionally much more frequent in this data set than other sources in this data set, the *AutoML* instance has to cope with highly imbalanced data.

Friedman: The Friedman data sets are artificially generated data sets [88] that contain linear and non-linear relations between the features and the output label. Further, it adds noise (e) to the output of the function in that the friedman function is defined as follows:

$$y = f(\overrightarrow{x}) = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + e$$

It thus contains five continuous features, where $X \in [0, 1]$. To test whether the underlying model is capable of selecting important features, other random features can be randomly added to the feature space X. The *OpenML* id 917 contains 1.000 instances and adds 20 random features to the relevant ones. The target is to determine if the label is under or above a given threshold.

PC4: This data set (*OpenML* id 1049) was proposed by Shirabad, Menzies, et al. [211] and contains data from earth orbiting satellites, where the task is to detect defective software. The data contains 1.458 instances and 38 features ranging from the number of lines and a number of operators to the percentage of comments within the software code. The features were defined to characterise code features associated with software quality objectively. As a result, the features are retrieved by the feature extractors proposed by [59].

Telescope: This data set (*OpenML* id 1120) simulates within a Monte Carlo process gamma particles in a ground-based Cherenkov telescope using imaging techniques. The task is to identify gamma-ray events hidden in charged cosmic ray background in Cherenkov telescopes. The available information consists of pulses left by incoming and simulated photons and allows the discrimination of the information caused by the gamma signals. It consists of 19.020 instances and 11 continuous features.

[1] www.openml.org, last accessed January 30, 2023

5.2.2 Preferences

In order to evaluated our approach for *AutoML*, we define in this section the set of metrics used within our system. Although we focus on classification tasks, we have used all available metrics from the *Scikit-learn* library [185] that support numerical inputs and also that go beyond the classification task. This is due to the reasoning that the underlying utility cannot be initially determined based on a metric and that this is originally unclear. We choose for the set of metrics $\mathbf{L}$ the following metrics and refer for an in-depth description to the *Scikit-learn* library. For classification, we choose the *Precision*, *Recall*, *F1* metrics and additional for regression the *MSE*, the *maximal error* and the *log loss*. Additionally, we use the *hamming loss*, which estimates the fraction of labels that are incorrectly predicted and the *jacquard score*, which is defined by the size of the intersection divided by the size of the union of two label sets. Denote that the direction for some metrics in $\mathbf{L}$ is distinct. Thus, the *Metric Learner's* task becomes more difficult in that important metrics may need to be weighted in the opposite direction of other metrics.

For the general setup we used *TPOT* [177] as *AutoML* component to (i) create pipeline configurations for the subsequent preference elicitation and (ii) to perform the optimisation process based on a learned metric $\mathcal{L}^*$. For each metric function $\mathcal{L}^{(i)} \in \mathbf{L}$ we fitted an *AutoML* instance for one hour and extracted all evaluated pipeline configurations. Since the execution of the first *AutoML* instances only serves to generate segments, whereby the evaluation time for each pipeline configuration depends on the size of the training data set X_{train}, we can set a tight time limit. Within one hour *TPOT* generated on a small data set, such as *OpenML* 38 up to 19452 different configurations and thus different segments. Considering larger data sets (*OpenML* 179) with a high number of instances *TPOT* still generated 2349 segments. In this case 2349 leads to $2.757.726$ possible segment pairs U^{pair}. Based on the generated segments and predefined preferences we evaluate in Section 5.3.1 the chosen learning to rank approach and in Section 5.3.2 its integration into a new AutoML instance.

5.3 Evaluation

This section evaluates the proposed *LTR* approach towards its application on *AutoML*. First, we evaluate in Section 5.3.1 the *Metric Learner* component and thus give answer to the question how a new target function $\mathcal{L}^*$ can be learned considering the performance metrics in Section 5.2.2. The second part of this evaluation concerns the overall system and the impact of the learned metric $\mathcal{L}^*$. We show that our approach is capable of taking preferences stated within the *Preference Interface* into account. Further, it suggests based on an integrated single-objective *AutoML* instance *ML* pipelines that can outperform *AutoML* instances trained on a static, predefined metric.

5.3.1 Metric Learner

Within the *Evaluation Generator* and the *Metric Learner* component in Sections 4.2.2 and 4.2.4 we propose to use different metrics as features to build a new metric function an end-user or an domain expert may want to pursue. However, to evaluate whether the proposed approach can learn a utility-based metric besides the performance of the underlying *RankNet LTR* learner, also the number of needed pairwise comparisons is an indicator for the *Metric Learners* capabilities to learn a utility.

For evaluation, we assumed three different utilities. First, we assumed that the utility follows a selective metric, the accuracy metric. Thus the task for the *RankNet* model becomes to select from a set of metrics $\mathbf{L}$ the accuracy metric. Within a linear combination, we assume that the utility follows a linear combination of

metrics, where each metric in $\mathbf{L}$ has with consideration of the direction of the metric a predefined and equally distributed weight. The F1 score shows whether the *RankNet* model can learn a metric that goes beyond a linear combination. To evaluate the *RankNet* model regarding the number of pairwise comparisons, we generate from the set of pipeline configurations for each data set $1,250$ segment pairs U^{pair}. This set is split into an 80/20 train-test-split, where up to $1,000$ segment pairs are used for training, and 250 pairs are used to evaluate the model. In Table 5.1, we present the achieved results of correct predicted rankings regarding different training data set sizes.

Utility	**OpenML ID**	**Training Size**				
		10	100	250	500	1000
Accuracy	38	0.913	0.963	0.972	0.975	0.977
	179	0.913	0.945	0.960	0.966	0.967
	772	0.963	0.956	0.966	0.975	0.977
	917	0.943	0.974	0.981	0.984	0.986
	1049	0.890	0.966	0.977	0.978	0.979
	1120	0.917	0.958	0.967	0.971	0.973
Linear Combination	38	0.896	0.935	0.935	0.935	0.935
	179	0.791	0.825	0.810	0.810	0.805
	772	0.593	0.687	0.888	0.933	0.937
	917	0.949	0.965	0.968	0.968	0.969
	1049	0.861	0.943	0.957	0.967	0.972
	1120	0.913	0.961	0.968	0.974	0.979
F1	38	0.937	0.972	0.971	0.977	0.978
	179	0.889	0.931	0.961	0.974	0.977
	772	0.876	0.966	0.972	0.972	0.974
	917	0.964	0.982	0.982	0.983	0.985
	1049	0.883	0.932	0.970	0.980	0.981
	1120	0.940	0.951	0.964	0.977	0.990

Table 5.1: RankNet - Percentage of correct predictions on test judgements [139]

In the following, we discuss, based on the results in Table 5.1, the results achieved for the different predefined utilities.

5.3.1.1 Accuracy

The aim of this experiment was to show, whether *RankNet* is capable to learn to select the set of metrics $\mathbf{L}$. We set the accuracy metric as a predefined metric and ranked $1,250$ segments based on the accuracy. The results show that *RankNet* is capable, accordingly to the accuracy metric, of ranking correctly 222 out of the 250 segment pairs held out for testing after only ten pairwise comparisons. Considering more than ten pairwise comparisons, the percentage of correctly ranked segment pairs increases. In this setting, it shows that *RankNet* can learn to select metrics within ten pairwise comparisons. However, increasing the number of judgements increases the percentage even more. Still, here the question is whether a percentage of at least 89% on the Friedman data set is sufficient to steer an *AutoML* instance.

5.3.1.2 Linear Combination

Within the second experiment, we tested whether *RankNet* is capable of learning a metric out of a linear combination. The sum of all weights is one. However, this task results in a more complex setting in that some metrics, e.g. *MSE*, are unbounded in that they possibly can get infinite. In Table 5.1, we see that in the case of a linear combination, the *RankNet* mode performs slightly worse than in the case of a selective metric. In general, this can be compensated by considering more pairwise comparisons. However, within ten pairwise comparisons, at least 59.3% (*OpenML* 772) are correctly ranked, and when considering 100 judgements, at least 68.7% pairwise judgements are correctly ranked on the test set.

5.3.1.3 F1

In this experiment, we show that our experiment can learn preferences that go beyond a linear combination. We remove from the set of metrics $\mathbf{L}$ the F1 metric and set it further as the metric that should be pursued. As shown in Section 2.6.1, the F1 score can be expressed by a combination of precision and recall. Assuming a F1 metric, the equation for the F score (Equation 2.25) can be reduced to

$$F1(\text{precision}, \text{recall}) = 2\frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{5.1}$$

By following the F1 metric, we show, on the one hand, similar to the selective case, that *RankNet* is capable of neglecting metrics and, on the other hand, that *RankNet* can learn complex correlations in the form of non-linearity. Table 5.1 shows that *RankNet* is capable of achieving at least a score of 87.6% on all data sets after 10 pairwise comparisons.

Summarising the capabilities of the *RankNet* model within the *metric learning* setting, this section clearly shows that it can learn an underlying metric in about ten pairwise comparisons. However, it is also necessary to point out the weaknesses of this approach. Within Section 5.2.2, we depicted the metrics that were limited to the performance metrics. Thus, the mapping range of the benefits is now also limited to the metrics in $\mathbf{L}$. Until now, we did not consider utilities such as the latency or the complexity of the *ML* pipeline, and thus these can not be mapped within the *Metric Learner* component. However, to answer RQ I.1, we showed in this evaluation that a *RankNet* model can learn a new metric within 10 to 100 comparisons considering a set of predefined metrics $\mathbf{L}$.

5.3.2 System Evaluation

Another important aspect is the impact of the learned metric on the performance of an *AutoML* regarding the predefined utilities. This section aims to evaluate this impact and thus answer RQ I.2. As depicted in Figure 5.1 and similar to Section 5.3.1, we created based on the metric functions in $\mathbf{L}$ various *ML* pipelines. Following Algorithm 3 within the *Evaluation Generator* component, we randomly created segment pairs and judged them based on the predefined utility (accuracy, linear combination and F1-score). The judgements are as stated in Section 5.3.1 used to train a *RankNet* model with the *Metric Learner* component.

In this section, we execute the *RankNet* models as utility-based metrics within an *AutoML* instance and measure the difference in performance and whether it leads to better performances regarding the predefined utility. To further compare the differences, we trained accordingly to the utility metrics for each metric a *AutoML* instance directly based on the predefined metric. We split each data set into a 80 / 20 train-test split and passed it together with the trained *RankNet* metric or directly with the assumed utility.

Utility	OpenML ID	AutoML Metric	
		Utility	RankNet
Accuracy	38	1.00	0.874
	179	0.710	0.611
	772	0.714	0.686
	917	1.00	0.984
	1049	0.774	0.618
	1120	0.922	0.899
Linear Combination	38	0.790	0.951
	179	2.207	1.100
	772	0.481	0.474
	917	4.861	3.395
	1049	6.261	2.023
	1120	1.692	1.067
F1	38	0.971	0.966
	179	0.706	0.700
	772	0.793	0.754
	917	1.000	0.978
	1049	0.978	0.733
	1120	0.984	0.963

Table 5.2: Evaluation utility-based *AutoML* [139]

In Table 5.2, we present the results achieved accordingly to the utility metric. Since we compare the approach with a *Metric Learner* component against *AutoML* instances directly fitted on the utility, the aim of this experiment is not to surpass the utility metric but to get as close as possible to the performance of the *AutoML* instance directly fitted on the utility. The scores presented for each utility were achieved on the 20% test split $\mathcal{D}_{test}$. In the following, we present as in Section 5.2.2 the evaluation for each predefined utility.

5.3.2.1 Accuracy

Considering the selective case, where we assume that the utility is represented within the accuracy metric, the *RankNet* model achieved after ten comparisons a score of at least 89.0%. However, for the overall system evaluation, it shows that in Table 5.2 *RankNet* manages to steer *AutoML* into the direction of the utility. In some cases (*OpenML* 38 and 1049) the *AutoML* instance with *RankNet* predicts significantly worse than the the *AutoML* instance directly fitted on the underlying utility. For the *OpenML* data sets 772 and 1120 the *AutoML* instance with *Metric Learner* component show, that *RankNet* is capable to steer the optimisation process closely into the direction of the utility. In average the *AutoML* instance with the *RankNet Metric Learner* component performance in average 7.4% worse than the instance directly fitted on the utility. These performance differences show preliminary the impact of choosing different metrics and that the approach with *Metric Learner* component is capable of steering *AutoML* optimisation processes in the direction of the utility.

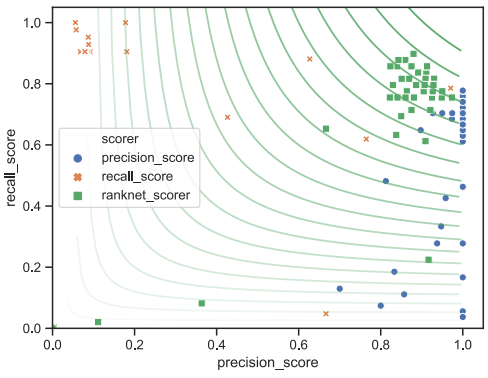
5.3.2.2 Linear Combination

Considering the case of a linear combination of metrics from the set $\mathbf{L}$, the task for the *RankNet* model becomes more difficult. Denote that the combination of metrics i.ex. the *MSE* is highly dependent on the data set and thus can possibly be infinite. However, Table 5.2 shows, that in the linear case the *AutoML* with the *Metric Learner* component is in some cases (*OpenML* 38) able to exceed the achieved performances of an *AutoML* instance directly fitted on the underlying metric. Since the range of this metric is undefined, the complexity of evaluating utility-based metrics becomes apparent. For example, the *MSE* score may have, due to its possibly higher value range than an accuracy metric, also higher importance when searching for suitable *ML* pipelines.

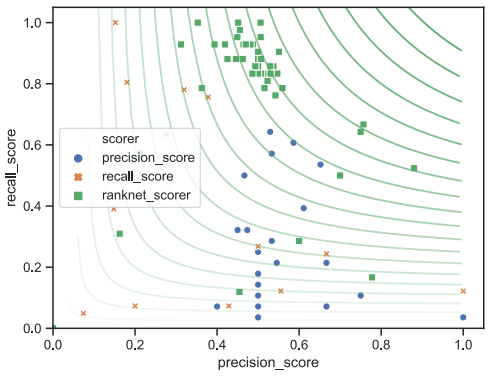
5.3.2.3 F1

In the last experiment, where we consider an F1 score as a utility to pursue, the task for the *RankNet* model was to neglect irrelevant metrics from $\mathbf{L}$ and to learn from the precision and recall metrics of the F1 metric score. Apparently, already in Section 5.3.1, we noticed that *RankNet* metric was capable to learn unexpectedly well this complex utility. However, this result is also reflected within Table 5.2, where the *RankNet* model is capable of steering the *AutoML* model in the direction of the utility and thus results in intimate performances to the *AutoML* instance directly fitted on the utility. On average the *AutoML* instance directly fitted on the predefined utility achieved 5.6% better performances than the *AutoML* instance fitted on the *Metric Learner* component.

In Table 5.2, we depicted the different performances of *AutoML* instance trained on the *Metric Learners* metric or directly on the underlying utility. However, the F1 score depends on precision and recall, which enables the visualisation of *ML* pipelines created by *AutoML* instances fitted on precision, recall and the learned F1 metric.



(a) AutoML instances on OpenML 38 - Precision, Recall curve [139]



(b) AutoML instances on OpenML 1049 - Precision, Recall curve [139]

Within the Figures 5.2a and 5.2b, we visualise the last 50 *ML* pipeline configurations created by *AutoML* instances following a precision, recall or learned F1 metric. The F1 metric is plotted as a function of precision and recall, where a higher colour intensity means a better F1 metric score. This experiment aims to show that the *Metric Learner* component is capable of steering the optimisation process of an *AutoML* instance in the direction of the *F1* score depending on Precision and Recall scores. Denote that the *RankNet* scorer does additionally learn from all metrics within $\mathbf{L}$ without considering the F1 score within

this set. In Figure 5.2a, we can see that the *AutoML* instances fitted on the precision or recall metric point in the direction of the respective metric. Further, it shows that the *RankNet* model can steer the *AutoML* optimisation process in the direction of the learned F1 score. While the *ML* pipeline configurations created by *AutoML* instances following the precision and recall scores optimise along the x and y-axis, the pipeline configurations optimised towards the learned F1 metric optimise towards the F1 metric. Thus, would an end-user's or domain expert's preference be the F1 metric but fits an *AutoML* instance based on the precision or recall metric, the *AutoML* instance would search for pipeline configurations that perform best on the x or y-axis. The *AutoML* instance fitted on the learned F1 score leads to better results than only fitting *AutoML* on the precision or recall metric when following the F1 score as a utility. The evaluation on *OpenML* 1049 depicted in Figure 5.2b demonstrates even clearer, that when an *AutoML* instance is trained on a metric an end-user or domain expert wants to pursue partially, the *AutoML* instance trained on a metric the user defined within pairwise judgements outperforms the other *AutoML* instances. Regarding RQ I.2, we showed, that the *Metric Learner* component is able to steer *AutoML* instances beyond a set of predefined metrics. This is particularly evident in evaluations depicted in Figures 5.2a and 5.2b.

5.4 Summary

In this chapter, we evaluated our approach proposed in Chapter 4 based on *AutoML*. We proposed a system that enables the integration of an end-users or domain experts utility and provided an empirical evaluation by assuming three synthetically generated utilities. Within the *Metric Learner* component, we integrated a *RankNet* model and showed that this model is capable of learning a new metric (RQ I.1) based on a few pairwise comparisons. Furthermore, the *Metric Learner* component steers an *AutoML* (RQ I.2) instance in the direction of the utility learned within the *RankNet* model. The RQ I.1 and RQ I.2 defined at the beginning of this work, can be answered within the evaluations in Sections 5.3.1 and 5.3.2 in that the *RankNet* model is able to learn a new target function $\mathcal{L}^*$ within a few pairwise comparisons and it steers the optimisation process of an *AutoML* system into the direction an end-user or domain expert may want to pursue. However, the limitations of our approach are the dependency on the set of already predefined metrics $\mathbf{L}$. It constrains the expressibility of the utility stated within the pairwise comparisons to this set. However, the set of available metrics can be extended with less effort. Further, we based our evaluation on randomly chosen pairwise comparisons. Here, the *Evaluation Generator* could improve the performance in that it actively selects pairs of segments where the *RankNet* model is uncertain.

6

Neural Architecture Search

This chapter provides an empirical evaluation of our approach toward the utility-based adaptation of *NAS*. In Chapter 5, we showed for *AutoML* that the general approach towards utility-based adaptation is capable of learning performance metrics by a few pairwise comparisons and thus that it is capable of steering the optimisation process of *AutoML*. However, as the underlying optimisation process is switched from a search towards the best suited *ML* pipeline to the best suited neural architecture, the search becomes more complex. The high variability and applications of *NN*s lead to enormous search spaces. Furthermore, the expensive computational training of *NN*s requires an efficient search in that only a few architectures are explored to find a suitable neural architecture. Even further, the high resource consumption implies that the underlying utility may not only rely on performance metrics such as the accuracy, precision but also on recall or the F1 score; It may incorporate metrics that go beyond the predictive performance and include, e.g. the latency of a *NN*. Thus in this chapter, we base our evaluation on metrics that go beyond the predictive performance and evaluate our approach against state-of-the-art multi-objective *NAS* approaches. As in Chapter 5, we provide in Section 6.1 a brief recap of the research questions, we aim to answer in this chapter. Further, we present in Section 6.2 an integrated utility-based *NAS* process, that considers the computational expensive evaluation of neural architectures and thus integrates an *ES* based *NAS* approach (following Real et al. [197] and Saltori et al. [205]) into the general utility-based system depicted in Figure 4.1 to reduce the number of needed evaluations. In Section 6.3, we depict the experimental setup by presenting the used data sets as well as the utility metrics that we aim to follow within the optimisation process. Finally, in Section 6.4, we evaluate whether the *Metric Learner* component is capable of learning a metric that goes beyond the predictive performance and whether the search for *NAS* system based on a learned metric is competitive against multi-objective *NAS* approaches. The results of this chapter have been published in "Personalized Neural Architecture Search" [140]. Furthermore, the implementations for the baselines as well an *OpenAI-Gym* environment for *NATS-Bench* [67] are made publicly available on *Github* [1].

6.1 Recap Research Questions

The search for suitable *NN*s is technically similar to the search for suitable *ML* pipelines. Further, they even incorporate similar *HPO* techniques. However, the architecture of *NN*s is diverse, complex and brings new challenges. As stated in the foundations (Chapter 2), *NAS* can take several days or weeks to find suitable neural architectures and thus opens a new dimensionality in complexity, compared to *AutoML* frameworks where the task is to find suitable *ML* pipelines. The optimisation process's emphasis may not lie in the predictive performance but in metrics that go beyond. While the main research question for *NAS* remains similar to the research question of utility-based *AutoML* systems, namely:

[1] https://github.com/kulbachcedric/, accessed on January 30, 2023

RQ II How can an *NAS* system be adapted to an end-user's needs?

The complexity as well as the possibilities of architectures and thus the enormous search spaces lead to the following sub research questions:

RQ II.1 How can a new target function $\mathcal{L}^*$ beyond predictive performance measurements be learned?

RQ II.2 How does a tailored target function influence *NAS* towards the pursued utility?

While for *AutoML* the research question was about the learning of a target function (RQ I.1, the first sub-question in *NAS* asks for a target function $\mathcal{L}^*$ that goes beyond the predictive performance measurements. An efficient search in *NAS* is crucial. Since an end-user or domain expert may not only retrieve a *NN* that has a good performance as well as a low latency, but also a suitable neural architecture that does not require training and testing the entire search space. Thus the impact of the learned metric needs to steer the optimisation process even better than in *AutoML*. RQ II.2 asks accordingly to the influence of the learned metric on the *NAS* optimisation process.

6.2 Integrated Utility-based Process

In this section, we present accordingly to Algorithm 2 an integrated approach towards utility-adaptation of *NAS*. This is motivated by the complexity of *NAS*. Thus, the aim is to minimise the number of *NN*s that need to be evaluated within the optimisation process. As in the *AutoML* setting, where the *Evaluation Initiator* component is part of a *ML* layer, that was not further addressed, in our approach the *Evaluation Initiator* as a single-objective *NAS* component part of a *NAS* layer (see Figure 6.1). As depicted in Section 4.2.1, the

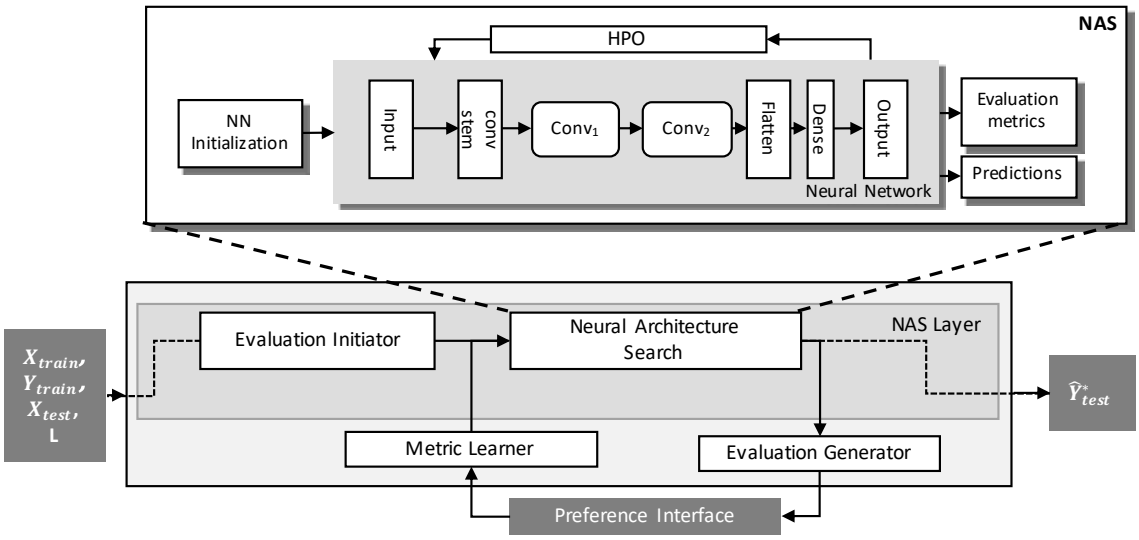


Figure 6.1: Utility-based *NAS* System

Evaluation Initiator generates a set of *ML* pipelines or neural architectures to create then a set of segments that are judged by an end-user or domain expert. However, considering an evolutionary approach, the initial generation of a population does not require an underlying metric. Thus, the generated set P can be used to (i) generate segments for the *Metric Learner* component and as (ii) initial population for the evolutionary *NAS* algorithm.

Algorithm 5 Utility-based *NAS*, following [197, 205]

1: **Input:**
2: Data set $\mathcal{D}_{\text{train}} = \{X_{\text{train}}, y_{\text{train}}\}$,
3: Data set $\mathcal{D}_{\text{valid}} = \{X_{\text{valid}}, y_{\text{valid}}\}$,
4: Set of Metrics $\mathbf{L}$,
5: Number of pairwise comparisons ω,
6: Population Size p,
7: Sample Size s,
8: Cycles c
9:
10: **Output:**
11: Neural Architecture $f^*_{g,\vec{z},\vec{\lambda}}$
12:
Ensure: $p \geq s \;\; \& \;\; c \geq p$
13: $P, H \leftarrow \emptyset$
14: **for** $i = 0$ to p **do** ▷ Generate Population P
15: $f_{g,\vec{z},\vec{\lambda}} \leftarrow RandomArch(G, \mathcal{Z}, \Lambda)$
16: $f_{g,\vec{z},\vec{\lambda}}.fit(X_{\text{train}}, y_{\text{train}})$ ▷ Train architecture
17: $P \leftarrow P \cup f_{g,\vec{z},\vec{\lambda}}$
18: $H \leftarrow H \cup f_{g,\vec{z},\vec{\lambda}}$
19: **end for**
20: $U^{\text{pair}} \leftarrow$ SegmentGenerator$(P, D_{\text{valid}}, \omega)$ ▷ see Algorithm 3
21: $U^{\text{judged}} \leftarrow$ PreferenceInterface(U^{pair}) ▷ see Section 4.2.3
22: $X^{(1)}, X^{(-1)} \leftarrow SampleComparisons(U^{\text{judged}})$
23: $\mathcal{L}^* \leftarrow$ RankNet.$fit(X^{(1)}, X^{(-1)})$ ▷ see Algorithm 4
24: **while** iter $\leq c$ **do** ▷ Start *NAS* iterations
25: $S \leftarrow$ sample s architectures $f^{(i)} \subset P$
26: $f^{(\text{parent})} \leftarrow \arg\max_{f \in S} \mathcal{L}^*(f(X_{valid}), y_{valid})$
27: $f^{(\text{child})} \leftarrow f^{(\text{parent})}$.mutate()
28: $f^{(\text{child})}.fit(X_{\text{train}}, y_{\text{train}})$
29: $P \leftarrow P \cup f^{(\text{child})}$
30: $H \leftarrow H \cup f^{(\text{child})}$
31: $P \leftarrow P \setminus P$.oldest
32: **end while**
33: $f^*_{g,\vec{z},\vec{\lambda}} \leftarrow \arg\min_{f_{g,\vec{z},\vec{\lambda}} \in H} \mathcal{L}^*(f_{g,\vec{z},\vec{\lambda}}(X_{valid}), y_{valid})$
34: **Return** $f^*_{g,\vec{z},\vec{\lambda}}$

Based on this idea, we depict in Algorithm 5 the overall process for an integrated utility-based adaptation of *NAS*. As *ES* we follow the regularised evolutionary single-objective *NAS* proposed by Real et al. [198]. In Algorithm 5, a population P of size p is randomly generated and trained in lines 13-19. Accordingly to Figure 6.1, in lines 20-23 the *Segment Generator* generates a set U^{pair} of segment pairs that are judged by the end-user or domain expert within the *Preference Interface*. The *Segment Generator* component uses the initially generated population P to create the set of segment pairs. In lines 22-23 of Algorithm 5, the *Metric Learner* component learns the underlying utility and passes it to the single-objective *NAS* optimiser. Within Algorithm 5, the regularised evolutionary algorithm proposed by Real et al. [198] uses in line 24 the initial population P and generates new populations by drawing individual architectures out of the population P, mutating and training them. In each cycle the oldest architecture is removed from the population. We denote the proposed approach in the following as regularised evolutionary algorithm.

6.3 Experimental Setup

In this section, we provide as for *AutoML* the experimental setup for an algorithm-centred evaluation. First, we present in Section 6.3.1 the data sets on which we evaluate our approach. We base our evaluation on the *NATS-Bench* [67] data set, that provides besides the data sets *CIFAR*-10, *CIFAR*-100 and *ImageNet*16-120 a unified search space. Further, it supports a broad range of underlying optimisation techniques, and in total, it contains $32.8k$ unique neural architectures distributed over the three data sets. *NATS-Bench* is split into a topology search space and a size search space, where the topology search space aims to represent different cell structures and the size search space seeks to optimise the number of cells in each layer. However, both search spaces are part of a fixed macro skeleton structure for the architecture candidates.

Considering a unified search space enables the comparison of different *NAS* approaches. Regarding RQ I.2 we integrated the regularised evolutionary algorithm, *LEMONADE* [73] and *MONAS* [114]. To measure the influence of a learned metric on *NAS*, we evaluate in Section 6.4.2 the regularised evolutionary algorithm, as well as *MONAS* with and without *Metric Learner*. In *MONAS*, Hsu et al. [114] considers a *RL* agent that learns to configure neural architectures based on a reward. This reward, however, is composed of different predefined metrics, and thus *MONAS* uses a single-objective optimiser (agent) to search for suitable architectures. It enables the integration of *MONAS* into our general utility-based approach. For this purpose and to foster the research on *RL* based optimisation techniques in *NAS*, we provide an *OpenAI-Gym* environment that contains the *NATS-Bench* search space as environment and returns based on a predefined metric $\mathcal{L}$ a reward. Furthermore, we use the predefined metrics proposed by Hsu et al. [114] depicted in Section 6.3.2 as utility for the evaluation of the *Metric Learner* component. *LEMONADE*, proposed by Elsken et al. [73] learns via Lamarckian evolutions a Pareto-frontier based on a set of metrics $\mathbf{L}$. Since it already optimises a Pareto-frontier, our *LTR* approach is not applicable. However, since it takes as our approach a set of metrics $\mathbf{L}$, it is comparable and thus part of our evaluation. Based on the unified *NATS-Bench* data set, we published the implementations of *LEMONADE*, the regularised evolutionary algorithm, *MONAS* and as well the *OpenAI-Gym* environment on *Github*.

In the following, we depict the data sets on which *NAS* is performed within the *NATS-Bench* data set and then the synthetic utilities accordingly to Hsu et al. [114].

6.3.1 Data Sets

For evaluation purposes we evaluate our approach based on the *NATS-Bench* [67] benchmarking data set. This technique has the advantage that we have access to a broad database of already trained *NN*s, and thus, it fosters research towards effective search strategies for *NAS*. In comparison to other *NAS* benchmark data sets such as *NAS-Bench-101* [247], *NAS-Bench-201* [68] and *NAS-Bench-301* [213], *NATS-Bench* contains besides the performance metrics fine grained information in different states of the training process of the neural architecture, further parameters such as the test/train accuracy and loss, the number of parameters and the latency.

The used data sets within *NATS-Bench* are presented in the following:

Cifar 10 The *CIFAR*-10 data set contains $60k$ colour images, where the task is to classify the images into 10 classes. The images have 32×32 pixels, where each pixel has three colour channels, and thus, an image contains $3,072$ features to classify it, ranging from aeroplanes to animals. It was collected by Krizhevsky [137] in 2009.

Cifar 100 Similar to the *CIFAR*-10 data set the *CIFAR*-100 data set has $60k$ colour images and was also collected by Krizhevsky [137]. However, it contains 100 classes (600 images per class) that can be grouped into 20 superclasses.

ImageNet 16-120 The ImageNet [62] data set contains $14M$ images that are annotated accordingly to the lexical database *WordNet* [168]. It contains $20,000$ different classes, whereby $\sim 1,2M$ images have bounding boxes and contain information about the location of the class within the image. In *NATS-Bench*, however, the images are downsampled to a size of 16×16 pixels (768 features) and filtered in that the images containing the first 120 classes ($151,7k$ images) are considered.

Comparing the data set with the data sets used within the evaluation of *AutoML* in Chapter 5, the image data sets contain significantly more features. This large number of image features is one reason why neural network training needs many computing resources. However, it emerges that the number of features and the complexity of *NN*s reinforce the necessity for an efficient optimisation strategy.

6.3.2 Metrics

We evaluate the *Metric Learner* component, as well as the performance of the state-of-the-art multi-objective *NAS* approaches in accordance with the metrics defined in "Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution" by Elsken et al. [73]. As in for the evaluation of *AutoML*, these metrics are divided into selective and combined metrics. Further, we normalise each metric $\mathcal{L}^{(i)} \in \mathbb{L}$ so that the point in the same direction and to $[0,1]$. Thus, higher return values for each $\mathcal{L}^{(i)} \in \mathbb{L}$ are better than lower return values.

Selective Metrics: The *NATS-Bench* benchmark data set provides for each *NN* the model's test/train accuracy and loss, the latency and the number of parameters are given. Considering a selective utility, one or few metrics $\mathcal{L}^{(i)} \in \mathbf{L}$ are pursued. The utility $h_{accuracy}$ (Equation 6.1) pursues the test accuracy, $h_{latency}$ (Equation 6.2) pursues the networks latency and h_{params} (Equation 6.3) follows the number of trained parameters [140].

$$h_{accuracy}(\mathbf{L}) = \mathcal{L}^{(\textit{accuracy test})} \tag{6.1}$$

$$h_{latency}(\mathbf{L}) = \mathcal{L}^{(latency)} \tag{6.2}$$

$$h_{params}(\mathbf{L}) = \mathcal{L}^{(params)} \tag{6.3}$$

Combined Metrics A combined utility can be a weighted metric as in the evaluation towards a utility-based *AutoML*, where each metric $\mathcal{L}^{(i)} \in \mathbf{L}$ is weighted by a parameter $\alpha^{(i)}$ (Equation 6.4). Further, we integrated constraint metrics, where the return is zero if the number of *FLOPS* or the accuracy exceeds a certain threshold. In Equation 6.5, we constrain the the *FLOPS* in that the metric returns the accuracy metric until the *FLOPS* do not exceed the median value of all neural architectures within the *NATS-Bench* data set. Further, in Equation 6.6, we constrain the accuracy in that the metric returns the *FLOPS* as values until the median accuracy of all neural architectures within the *NATS-Bench* data set is not exceeded.

$$h_{weighted}(\mathbf{L}) = \sum_{\mathcal{L}^{(i)} \in \mathbf{L}} \alpha^{(i)} \mathcal{L}^{(i)} \tag{6.4}$$

$$h_{\textit{flop constraint}}(\mathbf{L}) = \begin{cases} \mathcal{L}^{(accuracy)}, & \textit{if } \mathcal{L}^{(flop)} > threshold \\ 0, & \textit{else} \end{cases} \tag{6.5}$$

$$h_{\textit{accuracy constraint}}(\mathbf{L}) = \begin{cases} \mathcal{L}^{(flop)}, & \textit{if } \mathcal{L}^{(accuracy)} > threshold \\ 0, & \textit{else} \end{cases} \tag{6.6}$$

In addition to the evaluation towards utility-based *AutoML*, we add in the evaluation of an utility-based *NAS* system the assumption of constraint metrics. This has primarily a practical purpose. Since *NN* are relatively resource-intensive and in we want to provide a prediction in a limited amount of time and thus *FLOPS*, we consider that *NN*s that exceed a specific number of *FLOPS* are not applicable and thus have no utility.

6.4 Evaluation

In this section, we provide accordingly to the evaluation of *AutoML* an evaluation of the *Metric Learner* component in Section 6.4.1 that is sensitive to the number of pairwise judgments. The new requirements in *NAS* require as stated in Section 6.3.2 different metrics take the complexity of *NN*s into account. However, these metrics go beyond the predictive performance as evaluated in Chapter 5 for *AutoML* and may be constrained in that a certain threshold of *FLOPS* is not allowed to be exceeded or the end-user requires a *NN* with a minimum of accuracy. These new utility assumptions are evaluated in the Section 6.4.1 Further, we present in Section 6.4.2 the evaluation towards the impact of the *Metric Learner* component in comparison to other state-of-the-art multi-objective *NAS* approaches. While in *AutoML* we based the evaluation on one *AutoML* framework, namely *TPOT*, the research in *NAS* enables with benchmarks such as *NATS-Bench* [67] and the proposed approaches towards multi-objective *NAS*, the integration and comparison towards our general approach. This overall evaluation is presented in Section 6.4.2 based on the regularised evolutionary algorithm (see Algorithm 5 and [197]), *MONAS* [114] with and without *Metric Learner* component, as well as *LEMONADE* [73].

6.4.1 Metric Evaluation

To evaluate the *Metric Learner* component we randomly sampled 100 architectures $f_{g,\vec{z},\vec{\lambda}} \in P$ within the *Evaluation Initiator* component. The architectures are accordingly to Algorithm 3 processed to 1, 2, 3, 4, 5, 10, 25, 50 and 100 segment pairs U_{train}^{pair} within the *Evaluation Generator* component. As for the *Metric Learner* in *AutoML*, we use the predefined metrics from Section 6.3.2 to judge the segment pairs synthetically. Further, we repeat this process for all data sets within the *NATS-Bench* data set and all predefined metrics (Equation 6.1 - 6.6). In contrast to the evaluation of *AutoML* in Chapter 5, we evaluate applied the top-10 precision metric. This metric measures whether the *RankNet* model manages to rank the top 10 architectures out of 100 test architectures. Thus, a top-10 precision of 0.3 means that the ranking algorithm can recommend from the 100 generated test architectures three correctly within the top-10 architectures. In Table 5.1, we present the results for each data set and predefined utility accordingly to a variable number of training comparisons. It qualitatively shows that $\sim 10 - 25$ pairwise comparisons are needed to learn a utility an end-user or domain expert might pursue. However, Table 6.1 also shows that the constraint metrics (accuracy constraint and flop constraint) are more difficult to approximate. We assume that a rough knowledge of the underlying utility is necessary to steer the *NAS* optimisation process. Comparing the learning process considering a weighted utility of the *AutoML* approach, we note within the experiments of the *Metric Learner* component for *NAS* that it performs better than the constraint metrics. Furthermore, the selective metrics are well approximated after 10 pairwise comparisons, while the constraint metrics achieve their maxima in top-10 precision after ~ 50 comparisons. Concluding the evaluation of *Metric Learner* considering a *NAS* environment, the results are approximately the equivalent of those of

Table 6.1: RankNet - Top - 10 precision on test architectures

Data set	metric	train comparisons								
		1	2	3	4	5	10	25	50	100
Cifar-10	accuracy constraint	0,216	0,265	0,284	0,269	0,206	0,382	0,285	0,394	0,369
	accuracy	0,105	0,617	0,570	0,471	0,705	0,746	0,692	0,622	0,833
	flops constraint	0,282	0,215	0,288	0,168	0,234	0,241	0,232	0,294	0,419
	latency	0,713	0,696	0,512	0,770	0,703	0,753	0,659	0,671	0,665
	weighted	0,642	0,703	0,663	0,740	0,810	0,735	0,766	0,898	0,858
	params	0,321	0,312	0,428	0,418	0,308	0,682	0,577	0,539	0,565
Cifar-100	accuracy constraint	0,275	0,239	0,234	0,250	0,226	0,212	0,235	0,354	0,265
	accuracy	0,417	0,244	0,623	0,640	0,594	0,783	0,683	0,727	0,826
	flops constraint	0,296	0,206	0,236	0,199	0,236	0,236	0,194	0,297	0,265
	latency	0,539	0,527	0,494	0,611	0,557	0,569	0,572	0,600	0,566
	weighted	0,349	0,494	0,483	0,527	0,474	0,546	0,677	0,679	0,770
	params	0,227	0,369	0,434	0,324	0,400	0,610	0,509	0,630	0,527
ImageNet16-120	accuracy constraint	0,257	0,257	0,159	0,162	0,184	0,144	0,240	0,265	0,317
	accuracy	0,242	0,696	0,637	0,679	0,698	0,527	0,803	0,790	0,832
	flops constraint	0,217	0,213	0,226	0,227	0,119	0,232	0,275	0,337	0,326
	latency	0,391	0,310	0,323	0,340	0,322	0,148	0,191	0,268	0,421
	weighted	0,421	0,572	0,573	0,576	0,594	0,616	0,628	0,742	0,685
	params	0,331	0,489	0,351	0,545	0,444	0,688	0,571	0,759	0,604

AutoML. We can answer RQ II.1 in that the proposed *LTR* approach is capable of learning a new objective that goes beyond the predictive performance within ~ 20 pairwise judgements. In the following, we evaluate the impact of the learned utility on the *NAS* optimisation process.

6.4.2 System Evaluation

Following the evaluation of the *Metric Learner* component and to further reduce the amount of necessary *NN* evaluations, we assume that only a rough knowledge of the underlying utility is required to steer the *NAS* optimisation process. In the following evaluation, when training the *Metric Learner* component, we set the number of pairwise comparisons to 10. Thus, we set for the regularised evolutionary algorithm (see Algorithm 5) as well as for *LEMONADE* a population size $p = 10$ and a sampling size of $s = 5$. As stated in Section 6.3, we evaluate the regularised evolutionary algorithm and *MONAS* with and without *Metric Learner* component, whereby the regularised evolutionary approach without *Metric Learner* corresponds to the algorithm proposed by Real et al. [197]. The *MONAS* evaluation without *Metric Learner* component evaluated on the different metrics defined in Section 6.3.2 corresponds to the approach proposed by Hsu et al. [114]. Denote, that the results are due to the unified search space intra comparable but not comparable with results reclaimed in original publications [197] and [114]. Since *LEMONADE* searches a Pareto-frontier, we performed the search based on all metrics available within $\mathbf{L}$. We limited for each algorithm and data set the number of epochs to 100, whereby the generation of the population is included within the number of epochs. In Figure 6.2, we depict the the results of *LEMONADE*, as well as for *MONAS* and the regularised evolutionary algorithm with and without *Metric Learner* component when pursuing a selective accuracy metric. It shows that the pairwise ranking model can steer the population of the regularised evolutionary approach in the direction of the predefined utility. However, when evaluating the *NAS* algorithm without *Metric Learner* component, the regularised evolutionary approach achieves within 100 epochs a score of 99.7% on *CIFAR*-10, 99.6% on *CIFAR*-100 and 99.8% on *ImageNet*16-120 the best performances. Denote that the utility is based on normalised metrics. Thus, the scoring is relative to the best possible neural architecture and means that the *NAS* algorithm found an architecture that has 99.7% of the performance from the best architecture within the *NATS-Bench* data set. *LEMONADE* achieves with 99.4% on *CIFAR*-10, 96.6% on *CIFAR*-100 and 99.8% on *ImageNet*16-120 comparable results to the regularised evolutionary

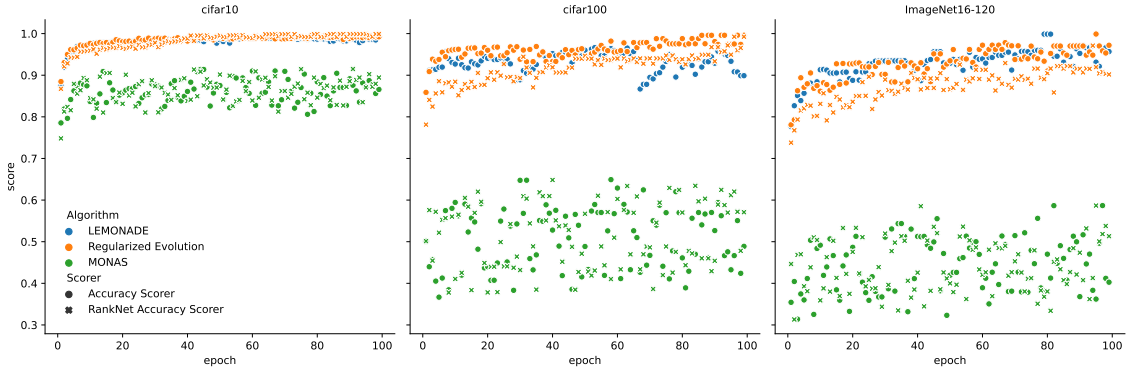


Figure 6.2: Utility scores based on a selective accuracy for *MONAS*, *LEMONADE* and the regularized evolutionary algorithm evaluated on *CIFAR*-10, *CIFAR*-100 and *ImageNet*16-120

approach. For *CIFAR*-100 and *ImageNet*16-120, *MONAS* performs significantly inferior and achieves 58.7% on *ImageNet*-120 and 64.9% on Cifar-100.

Considering the approach with *Metric Learner* component *MONAS* achieves 91.5% and the regularised evolutionary approach $99,8\%$ on *CIFAR*-10 within the selective accuracy utility setting. However, for *CIFAR*-10, as well as for *ImageNet*16-120, *MONAS* performs as well as in the setting without *Metric Learner* component significantly inferior (64.8% on *CIFAR*-100 and 58.8% on *ImageNet*16-120) considering the *NATS-Bench* search space. The regularised evolutionary approach achieves with *Metric Learner* similar results (99.2% on *CIFAR*-100 and 97.1% on *ImageNet*16-120) as without *Metric Learner* component. Since the difference between the approach with *Metric Learner* component and without *LTR* approach are marginal in both cases, *MONAS* and the regularised evolutionary algorithm, it shows that the *Metric Learner* component is capable to steer the optimisation process into the direction of the desired (selective accuracy) utility. Considering a weighted utility, where all metrics in $\mathbf{L}$ are equally weighted (Equation 6.4), the

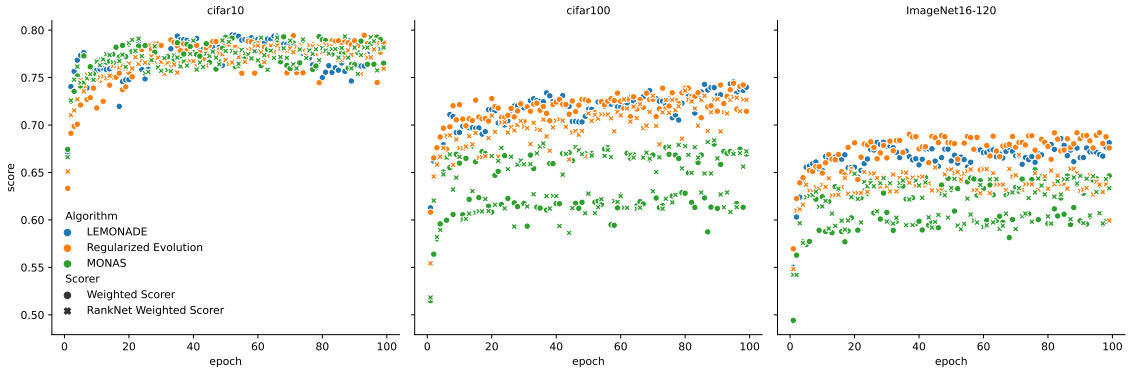


Figure 6.3: Utility scores based on an equally weighted preference for *MONAS*, *LEMONADE* and the regularized evolutionary algorithm evaluated on *CIFAR*-10, *CIFAR*-100 and *ImageNet*16-120 [140]

experiments show similar results. In Figure 6.3, we present the evaluation considering this weighted utility. For *CIFAR*-10, *MONAS* achieves a score of 79.3% and can compete against *LEMONADE* (79.5%) as well as the regularised evolutionary algorithm (79.4%). It is noticeable that the weighted utility is generally more complex to approximate than the selective accuracy utility. Both are due to the normalisation of all metrics in $\mathcal{L}$ comparable. The regularised evolutionary approach, as well as *LEMONADE*, perform within the 100 epochs similarly well. Furthermore, the differences between the approaches trained directly on the

underlying utility and trained with the *Metric Learner* component are negligible for *CIFAR*-10 and *CIFAR*-100. Differences coma apparent when training the regularised evolutionary approach on *ImageNet*16-120. Here, the regularised evolutionary approach achieves better results when directly trained on the predefined underlying metric. However, a central motivation was to empirically prove that the proposed *LTR* approach is capable of steering a *NAS* optimisation process (RQ II) in the direction of the utility an end-user or domain expert might pursue. Since the evaluation of the *Preference Executor* component has shown that pairwise comparisons can learn a suitable metric and thus lead to an appropriate recommendation of neural architectures (RQ II.1), we can answer the overall RQ II for *NAS* in that our proposed approach enables the steering of the optimisation process beyond predictive performance metrics.

6.5 Summary

In this chapter, we provided utility-based adaption of *NAS*. We empirically showed that *NAS* systems could be adapted to an end-user's or domain expert's utility that goes beyond predictive performance. We implemented state-of-the-art (multi-objective) approaches, as well as an *OpenAI-Gym* environment to evaluate various *NAS* approaches based on a unified benchmark search space (*NATS-Bench*). Our results for *CIFAR*-10, *CIFAR*-100, and *ImageNet*16-120 from the *NATS-Bench* data set are promising in that our *LTR* approach depicted in Chapter 4 is capable to steer the search process into the direction an end-user or domain expert might pursue. This, however, enables the consideration of key figures such as latency and *FLOPS* and thus an optimisation that searches in the direction of an underlying utility. In summary we can answer RQ II in that our *LTR* approach proposed in Chapter 4 is able to adapt to the utility of an end-user or domain expert.

However, since we provide an algorithm-centred evaluation towards a utility-based adaptation of *AutoML* and *NAS*, a human in the loop evaluation as often conducted in *HGML* is an exciting research field. However, these approaches are highly dependent on an adequate interface to allow the end-user to express their utility and thus represent a research field itself. As in Chapter 5 discussed, an active learning approach within the *Evaluation Generator* component, where the pairwise segments are chosen based on the expected information gain of a judgement, would possibly result in better steering of the underlying optimisation process.

Part IV

Stream Adaptation

7

Online Learning

In the previous chapters, we investigated the adaptation of a utility-based system in a batch learning setting. We focused on the beginning of a *DM* process, where the task is to understand the business and the data and thus define the underlying goal of the *DM* process. In this chapter, we investigate the continuous adaptation, where the underlying goal is already defined, but the data and the underlying patterns change over time. Thus, we investigate the evaluation and modelling steps at the end of the *DM* process. In Hypothesis II we assume that (i) the configuration of *AutoML* and *NAS* systems is according to the requirements defined by Bifet et al. [23] possible in an incremental manner and that the this adaption (ii) enables better performances in form of adaptation to potentially infinite data streams and changing patterns of *AutoML* and *NAS* systems.

Hypothesis II (Stream Adaptation)
In an online learning environment, the incremental adaptation of hyperparameters enables superior performance on data streams by aligning the learning process by following the online learning requirements defined by Bifet et al. [23].

To evaluate *AutoML* and *NAS* systems under the assumption of changing data patterns, we presented in Chapter 3 the related work for *online ensembles* and *online deep learning* approaches. Furthermore, we depicted in Section 2.5 the foundations for an adaptation to data streams and thus the concept of *online learning*. These foundations contained according to the *ML* pipeline for the *batch learning* case the adaptation of preprocessing steps as well as different established *online learning* models and approaches. Building on the foundations and the related work, we present in this chapter an overview over the frameworks that are developed and evaluated within Chapter 8 (Contribution II - Online Learning framework) and in Chapter 9 (Contribution III - Online Automated Machine learning Framework). Additionally, we present the data streams used to evaluate the capabilities toward stream adaptation and the general evaluation setup.

While frameworks such as *MOA* [25], *Scikit-Multiflow* [172] and *river* [173] enable the straightforward application of *online learning* algorithms, the research towards *online deep learning* techniques is as stated within the related work in Chapter 3 diverse and does not necessarily follow the *online learning* requirements defined by Bifet et al. [23]:

- *R I–1* Process an instance at a time and inspect it (at most) once.
- *R I–2* Use a limited amount of time to process each instance.
- *R I–3* Use a limited amount of memory.
- *R I–4* Be ready to give an answer (e.g. prediction) at any time.
- *R I–5* Adapt to temporal changes.

Furthermore, these frameworks incorporate building *ML* pipelines, but an automated pipeline configuration is not considered yet. The related work showed that *online learning* ensembles enable the exploitation of the different advantages of homogeneous and heterogeneous algorithms but do not incorporate an automated configuration during the data stream.

7.1 Frameworks for Online Analysis

In this section, we present an overview of existing *online learning* frameworks As stated within the foundations the application of *ML* models on data streams require a evaluation and monitoring of the model's performance, as well as preprocessing and *ML* algorithms, that are different from traditional batch learning. Frameworks with a growing community are *MOA*, *creme*, *Scikit-multiflow* and *river* that are presented in detail in the following:

MOA[1] stands for *Massive Online Analysis* and was developed at the University of Waikato by Bifet et al. [25] in 2010. It provides a broad collection of classification, regression, clustering, outlier detection and recommender system algorithms. It is written in *Java* and provides for evaluation purposes a Graphical User Interface as well as an *API* for further developments and extensions. The aim of *MOA* is to provide a benchmark suite for stream mining [25] and thus to foster research in the area of *online learning*.

Scikit-multiflow[2] was developed by Montiel et al. [172] in 2018 based on the ideas of *Scikit-learn* [185], *MOA* [25] and *MEKA* [196] as multi-label learning framework. As *MOA* it features a broad range of classification, regression and clustering algorithms. Further, it follows the *Scikit-learn* design principles (depicted in Section 7.2) and is designed to inter-operate with the Python *NumPy* and *SciPy* packages. It is implemented in Python follows a similar *API* to *Scikit-learn* and incorporates various evaluation protocols (see Section 2.6).

Creme[3] is another Python library for online *ML* that focusses on algorithms that can be updated with a single observation at a time (Requirement *R I–1*). While *Scikit-multiflow* [172] and *Scikit-learn* preliminary use *Numpy* arrays to interoperate with other frameworks or components, *creme* uses Python dictionaries. This is advantageous for interpretability since it enables to access features within a developed *ML* pipeline by their name.

River[4] emerged in 2021 by the fusion of *creme* and *Scikit-multiflow* and is actively developed and maintained by Montiel and Halford [173]. It compromises a the broad range of algorithms from *Scikit-multiflow* and the *API* from *creme*. While the aim of *MOA* is to provide a benchmark suite, the objective of *river* is to foster the applicability of streaming algorithms in real-world scenarios. Thus, the ability to orchestrate preprocessing and *ML* algorithms simplify the usage of developed algorithms. Like *Scikit-multiflow* and *creme* it is implemented in Python.

[1] `https://moa.cms.waikato.ac.nz/`, last accessed January 30, 2023
[2] `https://scikit-multiflow.github.io/`, last accessed January 30, 2023
[3] `https://github.com/MaxHalford/creme`, last accessed January 30, 2023

Since *river* is easy to extend and provides the capability of concatenating algorithms to *ML* pipelines, we base our implementations on *river* and extend it by an online *AutoML* and *DL* framework. Furthermore, it follows the *Scikit-learn* design principles, that we aim to base on our framework and on which are depicted in more detail in the following.

7.2 Scikit-learn Principles

With the development of *Scikit-learn* in 2011 by Pedregosa et al. [185], the application of (*offline*) *ML* algorithms became easy accessible. Nowadays, it is the most widely used Python library for *ML* and provides a vast number of algorithms to perform tasks such as classification, regression, but also clustering or dimensionality reduction. In *Scikit-learn*, each model is subject to the same *API*, which essentially fulfils five main design principles. One key driver for the success of this framework are these design principles, depicted in the following:

Requirements III. Scikit-Learn's design principles following [42, 185, 94]:

R III–1. All objects share a consistent and simple interface.

R III–2. All hyperparameters are directly accessible and exposed as public attributes.

R III–3. Algorithms are the only objects to be represented using custom classes. Data sets are represented as sparse matrices and hyperparameter names as well as their values are expressed as standard Python strings or numbers.

R III–4. Many *ML* tasks are expressible as sequences or combinations of transformations to data. Whenever feasible, algorithms are implemented and composed from existing building blocks.

R III–5. Whenever an operation requires a user-defined parameter, the library defines an appropriate default value.

Subsequently, we apply these requirements to the aforementioned *online learning* frameworks.

7.2.1 Consistency

The first requirement *R III–1* targets the consistency in that all objects share a consistent and simple interface. In *river*, as well as in *Scikit-learn* these are `Estimators`, `Transformers` and `Predictors`. Any object in *river* or *Scikit-learn* is based on an estimator, which it implements a `learn()` or `fit()` method that adapts the estimator to the learning setting. To emphasise the incremental training in *river* an estimator additionally implements a `learn_one()` method. Some estimators can also transform data and data streams. Typically these are transformers and used to preprocess the data according to the steps depicted in Section 2.2. In addition to the estimators they implement a `transform()` method that transforms a given input into a defined or learned output. The prediction (classification or regression) is executed within a predictor that is, as well as the transformer, an estimator. It implements, in addition to the estimator, a `predict()` method, which is used to return a class or a value based on the previously seen training instances. Considering our

[4] `https://riverml.xyz/latest/`, last accessed January 30, 2023

framework that integrates *DL* models, a *NN* (based on *Tensorflow* or *PyTorch*) can be integrated as predictor for both supervised- as well as unsupervised learning problems. Since the emphasis of this thesis lies within supervised learning problems, we confine the description of the framework to supervised learning, predominately classification problems.

7.2.2 Accessibility

The second requirement *R III–2* targets the accessibility of parameters of an estimator. To perform *HPO* and to understand the principles of operation, the parameters of the estimators need to be accessible. That incorporates getting and setting parameters in a uniform interface. In *river* this is achieved by passing a Python dictionary with key-value pairs to the estimator, which then works with the underlying model. This enables, in principle, the adaption of parameters while the model is consuming data from the underlying data stream. Since parameters, like the optimiser to be used, are often categorical, finding suitable values and adapting them during operation is another challenging tasks which we address in Chapter 9. For our frameworks, this design principle mainly considers simple access and configuration of the neural architecture as well as the underlying optimiser for the *DL* framework and an easy configuration of the *AutoML* estimator.

7.2.3 Classes

The third requirement *R III–3* targets the non-proliferation of classes. In *Scikit-learn* data sets are represented and exchanged in between the estimators as *NumPy* arrays. In *river* due to the nature of data streams, Python dictionaries are used to represent instances of the stream and to be exchanged between the estimators. Relying on rudimentary objects lowers the barrier of entry, avoids framework code and keeps the number of different objects to a minimum [185]. Furthermore, hyperparameters are regular strings or numbers, which eases the application of *HPO* techniques.

7.2.4 Composition

The fourth requirement *R III–4* targets the composition of estimators. Considering different estimators within *Scikit-learn* or *river*, *ML* pipelines are mapped by concatenating estimators. In *Scikit-learn* and *river*, these pipelines are mapped by an arbitrary created sequence of transformers followed by a final classification or regression estimator. As depicted in Section 2.2 the aim of *AutoML* is to automatically build *ML* pipelines. The composition of estimators enables building *ML* pipelines and is thus the basis for *AutoML*. The application of *NN*s as `Predictor` within *river* as part of our frameworks enables the concatenation of further estimators such as cleaning and normalisation steps are possible.

7.2.5 Default Variables

The last requirement *R III–5* targets the default parametrisation of an estimator. A reasonable default parametrisation enables a simple usage and a rapid development of *ML* pipelines. Furthermore, it reduces the complexity of the underlying model. This design principle is crucial for the application of *NN* in *online learning* considering the heterogeneous related work in this research field. Furthermore, established default values in *offline learning* may not be applicable in *online learning*, particularly not, when considering the *online learning* requirements (R I).

7.3 Framework Design Overview

As we aim according to Hypothesis II to perform *AutoML* and enable *NAS* on data streams, we depict in this section the overall framework and evaluation design. This design also reflects the structure and the ordering of the evaluation. As depicted in the previous section, we build our frameworks (*AutoML* and *DL*) on the *river* library. In Figure 7.1 we depict the framework dependencies, whereby the frameworks are depicted in more detail in Chapter 8 (*online DL*) and Chapter 9 (*AutoML* and *NAS*). Besides the algorithms that are categorised in `Transformers` and `Predictors`, the *river* library provides different data streams (depicted in Section 7.4) and evaluation protocols.

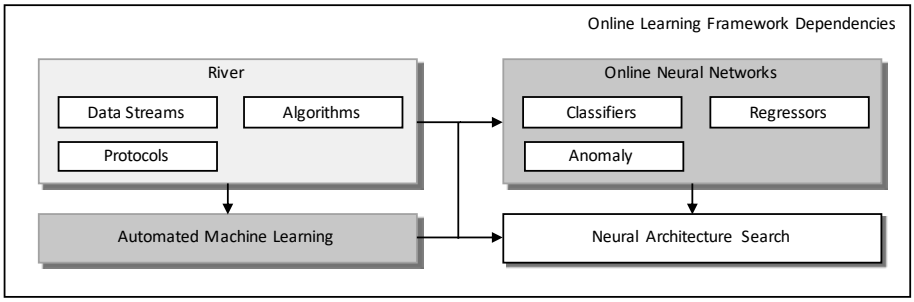


Figure 7.1: Illustration of the online learning framework dependencies.

In Figure 7.1 the boxes in dark grey are referred to as the frameworks that are part of the contributions of this thesis. While the *online DL* and the *AutoML* libraries are implemented as separate frameworks, *NAS* utilises the *online DL* framework as search space and the *AutoML* framework as underlying streaming *HPO* optimiser. The *online DL* framework is implemented separately to *river*, since *NN* and especially their architectures are diverse in their configuration opening a separate research field to *river*. In *batch learning Scikit-learn* incorporates only a small fraction of *NN*s (e.g. *logistic regression*) and frameworks such as *PyTorch* and *Tensorflow* allow the development of complex neural architectures. According to this development, we separate the development of *online NN*s from the *river* library. To implement neural architectures, we base our framework on *PyTorch* and follow the *river API*. This combination of both frameworks aims to simplify the application of *online learning* models within a unified stream processing *API* and to design complex neural architectures within the capabilities of *PyTorch*.

Accordingly *AutoML* systems in *batch learning* that rely on *Scikit-learn* (e.g. *autosklearn* [81, 80] or *TPOT* [177]) and to the development of *Scikit-learn* and *PyTorch* or *Tensorflow* we build our *online AutoML* framework separate from river, which it follows as the *online DL* library the *river API*.

7.4 Data Streams

In this section, we provide an overview over the data streams used to evaluate the *online DL* and the *AutoML* framework. In comparison to *offline learning* tasks, where huge data sets are collected and labelled (e.g. *CIFAR*), in *online learning* synthetically generated data streams are commonly applied to evaluate *online learning* algorithms. Furthermore, the generated streams have the advantage that they can be parametrised. However, besides these synthetically generated data streams, possibly infinite, some real-world data sets are iteratively processed and thus used as data streams. These data streams are depicted in the following, whereby their configuration is part of the experimental setup of the respective evaluation.

7.4.1 Real-world Streams

Real-world data sets that are commonly applied in *online learning* to evaluate new algorithms are preliminary the Covertype and the Electricity data set. Both are considered can be depicted as follows:

Covertype [28] The Covertype data set contains forest cover type information collected by the US Forest Service. It contains 581,012 samples with 54 attributes ranging from binary variables such as the soil to continuous values such as the elevation of $30m \times 30m$ cells. The task is to classify the cell into one of 7 possible cover type classes. In *online learning* applications, the data set is iteratively processed and contains dependencies within the order of the input sequence [27].

Electricity [25] The electricity data set with $45,312$ instances, is a relatively small set that is used to evaluate algorithms on data streams. It contains six continuous features of the Australian New South Wales Electricity Market. The goal is to identify the electricity price change relative to the moving average of the last 24 hours. The data stream is thus a binary classification task. As the electricity market price evolves, it has a time dependency and can, like, the Covertype data stream, be seen as multivariate time series.

7.4.2 Synthetic Streams

In contrast to synthetic data streams, real-world data streams have the advantage that they can be parametrised and assume a particular function or distribution to be learned. Furthermore, they are possibly infinite, and by changing the underlying distribution or the pattern, concept drifts can be implemented artificially at a point in time or in a time range. Due to their infinite nature, they are also referred to as generators.

Agrawal [2]: The Agrawal generator was proposed by Agrawal et al. [2] in 1993. Based on 6 numerical and 3 categorical features that compromise e.g. salary, age or education level the task is to predict whether a loan will be approved or not. This ground truth of this binary classification is based on ten predefined functions. The generator calculates attribute-values according to value-ranges appropriate for the respective features. A concept drift can be implemented within the stream by changing the underlying classification function.

***SEA* [219]:** The *Streaming Ensemble Algorithm* contains 3 features ranging between 0 and 10. Two of the features are relevant and classify the features as true if:

Variant 1: $f(\overrightarrow{x}) = x_1 + x_2 > 8$

Variant 2: $f(\overrightarrow{x}) = x_1 + x_2 > 9$

Variant 3: $f(\overrightarrow{x}) = x_1 + x_2 > 7$

Variant 4: $f(\overrightarrow{x}) = x_1 + x_2 > 9.5$

The third feature is irrelevant. A concept drift can be implemented by switching the variant during the data stream.

Hyperplane [116]: The Hyperplane data stream creates as the name states a hyperplane that splits an n-dimensional space and thus classifies n features according to if they are above or under the hyperplane. The hyperplane is defined by:

$$\sum_{i=1}^{d} w_i x_i = w_0 \tag{7.1}$$

This generator supports setting the number of features, whereby all features are relevant. Furthermore, drifts can be implemented by changing a predefined number of features from the feature set according to a preset magnitude of change. The features ranges from 0 to 1.

LED [98]: This generator produces a stream with 24 binary attributes of which 17 are irrelevant for classification. The goal is to predict a digit displayed on an LED panel with seven segments. Thus it contains ten different classes, classified by seven binary labels. The features are randomly created, whereby each attribute has a 10% chance of being inverted. The structure of this data stream does not allow for synthetically inserted concept drifts.

Sine [91]: The Sine generator is likewise the Hyperplane generator a binary classification stream that classifies whether a generated feature set is above or under a certain function. However, the stream consists of 4 features, where only two are relevant and classified as true if:

Variant 1: $f(\overrightarrow{x}) < \sin(x_1) - x_2$

Variant 2: $f(\overrightarrow{x}) \geq \sin(x_1) - x_2$

Variant 3: $f(\overrightarrow{x}) < x_1 - 0.5 - 0.3\sin(3\pi x_2)$

Variant 4: $f(\overrightarrow{x}) \geq x_1 - 0.5 - 0.3\sin(3\pi x_2)$

A concept drift can be introduced by changing the underlying classification variant.

Random *RBF*: The Random *RBF* produces a radial basis function by a number of centroids. Each centroid has a random initial position and a weight. Every new instance is assigned randomly to one of the centroids according to its weight. A concept drift is implemented by randomly shifting the initial position of a centroid by a Gaussian distributed length.

While the *RBF* and the Hyperplane generator support continuous concept drifts, generators with different variants can change their distribution at a certain point. To detach this binding to a certain point p by a change width w a concept drift can be introduced by following a sigmoid function:

$$f(t) = \frac{1}{1 + e^{-4\frac{t-p}{w}}} \tag{7.2}$$

The sigmoid function defines the probability that each new instance x_t of the stream belongs to the new concept after the drift.

8

Online Deep Learning Framework

To foster research in *online deep learning* in a unified and reproducible form we depict in this chapter an *online deep learning* framework that builds on *river* [172] as *online learning* and *PyTorch* [184] as commonly used *DL* library. Furthermore, it follows the *Scikit-Learn's* design principles, depicted in Section 7.2, that have proven to be the key drivers of a broad adoption by the *ML* community. Within the foundations (Chapter 2), Section 2.5.4.3 briefly addressed the theoretic applicability of *NN*s in *online learning*. Further, the related work presented in Section 3.4, showed that *NN*s are applied in an *online learning* environment in a broad range of applications (see. Table 3.4). However, these approaches are insular in that they are limited to their domain and developed isolated in their application to other approaches. As the applicability of *NN*s was already addressed the question of suitability of *NN*s regarding the requirements of *online learning* remains open:

RQ III.1 Are neural networks suitable for online learning?

Locating the application of *NN* on data streams in the context of this thesis towards the adaptivity of *AutoML* and *NAS* techniques; this research question is fundamental for the application of *NAS* techniques. However, the aim of this chapter is to answer the suitability of *NN* regarding the requirements defined by Bifet et al. [23] which results in an *online DL* framework published on Github [1]. Therefore, we depict in Section 8.1 the underlying mode of function of the proposed framework by also considering the *Scikit-learn's* design principles. In Section 8.2, we provide the experimental setup towards an evaluation that shows that *NN*s are suitable for *online learning*. One requirement of the *Scikit-learn's* design principles is to provide reasonable default values (Requirement *R III–5*) for all algorithms. To answer research question RQ III.1 based on a suitable parametrisation, we evaluate our approach against different single algorithms and present the results in Section 8.3. The idea for this framework emerged within a research stay at Télécom ParisTech together with Professor Albert Bifet, Jacob Montiel and Maroua Bahri. It predominately aims to foster research in *DL* techniques for *online learning*.

8.1 Approach

In this section we depict the approach towards an *online DL* framework developed in *Python*. While the frameworks *MOA* [25], *Scikit-Multiflow* [172] and *river* [173] simplify the application of *online learning* algorithms, frameworks such as *PyTorch* [184] or *Tensorflow* [1] provide large capabilities in designing and executing deep learning models based on a given data set. In *online Learning*, however, *river* is becoming the leading Python platform for *ML* applications on data streams. Further, most of the design principles

[1] https://github.com/kulbachcedric/, last accessed on January 30, 2023

of *Scikit-learn* are adopted in *river* in that it is developed according to the Requirements III depicted in Section 7.2.

In *offline learning*, *PyTorch* developed by Paszke et al. [184] in 2016 and *Tensorflow* Abadi et al. [1] are two broadly used open source Python frameworks for *DL*. Both rudimentary support online learning techniques, but they are designed for batch learning. When considering e.g. requirement *R III–5*, the default parameters in a *offline learning* may not be suitable to *online learning*. For instance, the sequential processing of instances within the stream or the adaptation to concept drifts may require a higher learning rate. We base our implementation on *PyTorch*, as it is capable to dynamically change neural architectures while it is executed and it relies on *Torch*, which aligns with *R III–1* as lightweight interface. In comparison, *Tensorflow* requires a defined Graph structure at the beginning of the data stream and wraps each instance of the data stream into a Tensor class. Thus, *PyTorch* is capable to processes data in form of streams in comparison to *Tensorflow* faster (*R I–2*) and with less memory (*R I–3*) consumption as it does not require to wrap each instance into a separate class. Further, the related work that proposes optimisers or architectures that are specifically designed to adapt to data streams (e.g. [204]) may not be applicable in *offline learning* and thus are often not considered in these frameworks. Thus, from a technical point of view, another aim of our framework is to merge the stream focussed *river* framework with the frameworks that allow an eased implementation of *NN*s. In the following, we depict the merge of *PyTorch* with the *river API* within a separated framework. The integration of *DL* models is achieved by implementing compatibility wrappers as *river* `Predictors` for *PyTorch*.

This integration has the advantage of merging the large capabilities of *river* in that we follow the consistency requirement *R III–1* and enable the composition *R III–4* of already existing estimators within the *river* environment out of the box. Furthermore, it opens the broad capabilities of designing *NN* in established frameworks and thus contributes to the non-proliferation of classes *R III–3*. In Listing 8.1, we depict the integration of a logistic regression using *PyTorch* as underlying *DL* library.

Listing 8.1: Application classification model

```
from river import compat
from river import datasets
from torch import nn
from torch import optim

def build_torch_logistic_regression(n_features):
        net = nn.Sequential(
                nn.Linear(n_features, 1),
                nn.Sigmoid()
        )
        return net

model = compat.PyTorch2RiverClassifier(
        build_fn= build_logistic_regression,
    loss_fn='bce',
    optimizer_fn='sgd',
    learning_rate=1e-3
    )
```

In Listing 8.1 it emerges, that the wrapper class `PyTorch2RiverClassifier` is imported in line 2 from the *river* framework and thus is part of the *river* framework. As *Scikit-learn* integrates basic functionalities for *DL*, we integrated the `PyTorch2RiverClassifier` into the *river* framework. This allows according to the *Scikit-learn* design principles basic functionalities in implementing *NN*s. However, for the development

of a broad variability of *NN*s, as well as to build a library that includes architecture solutions for a wide range of applications, we created a separate framework. This framework includes beside the wrapper functionalities of *PyTorch* into *river* custom optimisers and architecture modules (e.g. *LSTM*) to perform not only supervised classification and regression tasks, but also to perform *anomaly detection*. To meet the requirement of a simple interface (*R III–1*), we integrated the wrappers that enable rudimental *FNN* neural architectures into the *river* framework and refer to our framework for the application of sophisticated *NN*s. To avoid losing the scope of this work, we will rely in this thesis on the classification and regression task for *NN*s that enable the application of *NAS* on data streams.

8.1.1 Configuration

Listing 8.1 depicts the configuration of the wrapper predictor to execute it within *river*. While parameters are configured as strings or numbers, the neural architecture is passed as a function into the wrapper class. These architectures can range from simple linear logistic regression or *MLP* to complex *CNN* and *RNN* architectures. Passing a function instead of a string or number violates requirement *R III–2*; however, due to simplicity and in accordance with an existing (*batch learning*) *Tensorflow* wrapper within the *Scikit-learn* framework, we define a neural architecture within a function and pass this function into the wrapper function. Note that other parameters are set by the requirements *R III–2*. Additional features for the neural architecture required within the initialisation can be specified as a function parameter of the *NN* build function (see Listing 8.1) and passed to the wrapper class within the initialisation. To automatically set the number of features that the data stream contains, the *NN* build function includes a *n_features* parameter that is set within the first instance of the data stream when the *NN* is initialised within the wrapper class. In the following we depict the training process as part of the `learn_one()` function from the *river API*.

8.1.2 Training Process

The general training process for *NN*s on data streams, considering a (gradient-based) back-propagation technique, iteratively updates the weights of a neural architecture by predicting a label for each instance of the data stream and performing on the prediction made an optimisation step. In Algorithm 6, we depict this general process and consider that additionally to the neural model and the data stream, an optimiser $optim$ (e.g. *SGD* or *Adam*) as well as a loss metric $\mathcal{L}$ are given. Furthermore, we consider that the optimiser is preconfigured in that parameters such as a learning rate are already parametrised within the initialisation of the `PyTorch2RiverClassifier` class. The back-propagation steps are applied based on on a prediction of the features $\overrightarrow{x}_t$ that are part of a streaming instance e_t.

Algorithm 6 Back-propagation on Data Streams

Input:
Data stream $\mathcal{S}$,
Neural model $f_{g,\overrightarrow{z},\overrightarrow{\lambda}}$,
Loss metric $\mathcal{L}$,
Optimiser $optim$

if e_t **then** ▷ Start Data Stream
    $\hat{y} \leftarrow f_{g,\overrightarrow{z},\overrightarrow{\lambda}}(x_t)$
    $\overrightarrow{\lambda}^* \leftarrow optim.optimise(\overrightarrow{\lambda}, \mathcal{L}(y_t, \hat{y}))$
    $f_{g,\overrightarrow{z},\overrightarrow{\lambda}} \leftarrow f_{g,\overrightarrow{z},\overrightarrow{\lambda}^*}$
end if

Considering a streaming based back-propagation enables to process an instance at a time (Requirement *R I–1*), however, this comes with the cost of performance. In *offline learning* all data instances are available at a time in form of a data set $\mathcal{D}$. Processing data in a batch, however, enables an efficient and parallel processing of large matrices within the optimizer $optim$ that is often accelerated by the use of *GPU*s or *TPU*s. The iterative processing when considering data streams prevents the calculation of data batches and thus the acceleration of *GPU*s and *TPU*s. Furthermore, the iterative processing might have a significant impact on the models performance. When updating the weight according to a over the data instances leveraged gradient (batch), the model might generalise better, since it considers more data points at once. Updating the weights of a *NN* iteratively with the data stream has in consequence that e.g. a too small learning rate does lead to convergence with the underlying distribution or a too slow adaptation to concept drifts and a too high learning rate leads to too large weight updates and thus to a non learning *NN*. Another challenge is the underlying optimiser, as data streams often occur in real-world applications with time dependencies, an optimiser that considers the momentum of the data stream might be beneficial towards the evaluation.

While Algorithm 6 is applicable on tasks where the output might not change over time, e.g. regression, for classification, the number of classes may change over time. For example, the first instance of a data stream only considers the information for the label y_1 of one class. The second instance may consider the information for a second class or may even have the same label as y_1. Thus, at the beginning of the data stream, the number of class labels is unknown, which means for a *NN* that, the number of outputs needs to be adapted over time.

(a) *NN* with two outputs p_1 and p_2 before a new class occurs within the data stream

(b) Adapted *NN* with averaged weight initialisation after new classes occurred within the data stream.

Figure 8.1: Schematic view for the adaptation of the last layer of an *NN* to new classes that occur within the data stream. $p_1, \ldots, p_{|classes|}$ indicate the output of the last layer and thus the probability for the classes $1, \ldots, |classes|$. The grey arrows denote the weights of the already existing *NN* and the black arrows denote the new weights initialised by averaging the old weights (see. Algorithm 7) of the algorithm.

Considering the case, where the number of labels or classes is unknown, we provide a class adaptive wrapper in Algorithm 7.

Algorithm 7 Class Adaptive Back-propagation on Data Streams

1: **Input:**
2: Data stream $\mathcal{S}$,
3: Neural model $f_{g,\overrightarrow{z},\overrightarrow{\lambda}}$,
4: Loss metric $\mathcal{L}$,
5: Optimiser $optim$
6:
7: $classes \leftarrow \emptyset$ ▷ Set of classes
8: **if** e_t **then** ▷ Start Data Stream
9: $classes \leftarrow y_t$
10: **while** $|classes| > |f(\overrightarrow{x})|$ **do**
11: $\overrightarrow{w}^{(out)} \leftarrow f.get\_layer(-1)$ ▷ Weights of last layer
12: $\mu_w \leftarrow mean(\overrightarrow{w}^{(out)})$
13: $\overrightarrow{w}^{(out)}.append(\mu_w)$
14: $f.set\_layer(-1, \overrightarrow{w}^{(out)})$
15: **end while**
16: $\hat{y} \leftarrow f_{g,\overrightarrow{z},\overrightarrow{\lambda}}(x_t)$ ▷ See Algorithm 6
17: $\overrightarrow{\lambda}^* \leftarrow optim.optimise(\overrightarrow{\lambda}, \mathcal{L}(y_t, \hat{y}))$
18: $f_{g,\overrightarrow{z},\overrightarrow{\lambda}} \leftarrow f_{g,\overrightarrow{z},\overrightarrow{\lambda}^*}$
19: **end if**

In Algorithm 7 the class adaption is applied in lines 9ff and further illustrated in Figure 8.1 In accordance to the data stream the classes, that occur within the stream are tracked within the $classes$ set. When the number of classes ($|classes|$) is higher than the number of output nodes of the underlying *NN* f, then the class adaption is applied in lines 11-14. In Algorithm 7 this is determined by a forward pass through the model ($f(\overrightarrow{x_t})$), but can in practice efficiently be called by the current underlying architecture at a time step t. The class adaption step takes the last trainable layer of the model ($f.get\_layer(-1)$) and appends a new (linear) node to this layer by leveraging the weights of the previously trained weights from the last layer (line 12) and appending it to the weights of the last layer. The new weights $\overrightarrow{w}^{(out)}$ replace with an additional output node the last trainable layer of $f_{g,\overrightarrow{z},\overrightarrow{\lambda}}$. By leveraging the weights of the last layer to initialise the additional node for the new occurring class within the data stream, we assume that the initial new class's probability is equal to the average of all other classes in this last layer. Whereby according to the instance e_t all weights of the new *NN* are updated by Algorithm 8.

8.1.3 Prediction Process

As stated in the training process, the model performs predictions according to the *NN*'s output. Considering a regression task, where the output of the model is fixed by the first instance of the data stream, the *NN*'s width of the output layer can also be set by the first instance of the stream. The value based on the forward pass on $\overrightarrow{x}_t$ can be used as the regression value for the prediction. In the case of classification, the output of the *NN*'s forward pass is used as a probability for each class. Denote that the sum of all layer outputs is not necessarily equal to 1 as it depends on the chosen activation layer.

In this section, we presented the extension of a well-known machine learning framework, *river*, to train neural networks in an online setting. It is the foundation for performing online *NAS*. We presented the consistent and simple *API* (*R III–1* and *R III–2*) according to the *river* framework that enables the concatenation with other estimators from this framework (*R III–4*). However, a contribution, which was developed during the process of this thesis, is a separate *online learning* framework for *DL*, mainly based on *PyTorch* that simplifies and unifies the execution of *DL* models in an *online learning* setting. This incorporates a broad

variability of predictors and transformers (e.g. auto-encoders), as well as predefined neural architectures. The framework aims to make *DL* approaches that prevail easy to apply and bring the success that *NN* have in batch learning to *online learning*. Furthermore, this framework is intended to simplify and unify research in this area. To define appropriate default values and thus investigate the requirement *R III–5*, we provide in the following an empirical evaluation of the learning rate as well as a suitable optimiser.

8.2 Experimental Setup

In the previous section we preliminary investigated the requirements *R III–1* to *R III–4*. In this section we investigate Requirement *R III–5* as well as a general evaluation towards the suitability of *NN* (i.ex. *MLP*) in *online learning*. While *PyTorch* provides reasonable default parametrisations, as well as established neural architectures in the case of batch learning, they might not be appropriate for *online learning*. When assuming an *online learning* scenario, a further challenge is an adaptation to concept drifts, where a too small learning rate leads to non-convergence with the underlying distribution or a too slow adaptation to the concept drifts. However, a too high learning rate might lead to over sensibility regarding an occurring concept drift. Another challenge in *online DL* is the underlying optimiser; as streams often occur in real-world applications with time dependencies, an optimiser that considers the momentum of the data stream might be beneficial towards the application of *NN*. Thus, a reasonable parametrisation in *online learning* is crucial. Therefore we present in the following an empirical evaluation for the (i) learning rate as well as the underlying (ii) optimiser. Note that these values can be set in addition to the search for a suitable architecture as a hyperparameter within *NAS*. According to Algorithm 6, the learning rate and the underlying optimiser are preset when applying *NN*s on data streams and thus require a reasonable default parametrisation when following requirement *R III–5*.

First, we depict the parametrisation of the used data streams in Section 7.4. We then present in Section 8.2.2 the experimental setup towards a reasonable default parametrisation, which also shows the first indications of the suitability of *NN* in *online learning*. This experimental setup includes the evaluation of a broad range of learning rates and different available optimisers. Based on the estimated default values, we evaluate different neural architectures against a selection of algorithms implemented within the *river* library. We executed all experiments on an *Intel(R) Xeon(R) Platinum 8180M CPU* with 2.50 GHz base clock and 1.5 terabytes of RAM without consideration of further *GPU* acceleration.

8.2.1 Data Streams

We base our evaluations in this chapter on a subset of the data streams depicted in Section 7.4. The configuration of the data stream generators is presented in Table 8.1. In Table 8.1, we depict the configurations of the data stream generators. The *SEA* generator is implemented with three drifts considering a predefined drift width of w. The first drift occurs at $1/4$, the second at $1/2$ and the third at $3/4$ of the data stream. The Agrawal stream is configured considering one drift at half of the stream and is configured likewise the *SEA* generator by a drift width of w.

Table 8.1: Configuration of Data Stream Generators with Concept Drifts

Data Stream	Setting
***SEA* Generator**	3 Features, 2 classes
Drifts	three drifts are assumed with a change width of w, denoted as *SEA*(w)
	First Drift:
	Drift Position: $1/4$ of instances
	Original Drift: *SEA* Variant 2
	Drift Stream: *SEA* Variant 3
	Second Drift:
	Drift Position: $1/2$ of instances
	Original Drift: *SEA* Variant 3
	Drift Stream: *SEA* Variant 4
	Third Drift:
	Drift Position: $3/4$ of instances
	Original Drift: *SEA* Variant 4
	Drift Stream: *SEA* Variant 0
Agrawal Generator	9 Features, 2 classes
Drift	one drift is assumed with a change width of w, denoted as Agrawal(w)
	Drift:
	Drift Position: $1/2$ of instances
	Original Drift: Agrawal Variant 1
	Drift Stream: Agrawal Variant 5
Random *RBF* Generator	10 Features, 2 classes, a centroids, denoted as *RBF*(a, b)
Drift	continuous change magnitude of b, whereby half of the centroids are considered to change within the drift
Hyperplane Generator	a Features, 2 classes, denoted as HYP(a, b)
Drift	continuous change magnitude of b

The Random *RBF* generator considers ten features and is configured as a binary classification stream, where the magnitude of change and the number of centroids are kept variable. The Hyperplane generator is configured with different magnitudes of change. Within the experiments, we set the number of centroids of the *RBF* generator to 10 and the number of features of the hyperplane generator to 10.

Besides the configurable generators, we evaluate our framework on the non-configurable synthetic streams LED and Sine and the real-world data sets Covertype and Electricity. Each experiment is evaluated on a stream with a length of $100,000$ instances.

8.2.2 Default Parametrisation

With regard to the framework's default parametrisation, we consider the learning rate and the underlying optimiser as parameters that are configured besides the neural architecture within the `PyTorch2RiverClassifier`.

Table 8.2: Experimental Setup Default Parametrisation

Parameter	Setting
Stream Generators	Agrawal$(50,000)$, Agrawal(50), Covertype, Electricity, HYP$(50, 0.0001)$, HYP$(50, 0.001)$, LED, RBF$(10, 0.0001)$, RBF$(10, 0.001)$, *SEA*(50000), *SEA*(50), Sine
Metric	Rolling Accuracy with a window size of $1,000$ instances
Preprocessing	Standard Scaling
Models	River Logistic Regression (only learning rate), Torch Logistic Regression, Torch Static *MLP*, Torch Dynamic *MLP*
Experiment Configurations	
Learning Rate	optimizer: *SGD*
	learning rates: 10^{-9}, 10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 25, 50, 100
Optimiser	optimizer: *SGD*, *SGDHD*, *RMSProp*, *Adam*, *AdamW*
	learning rates: 0.01

In Table 8.2, we depict the experimental setup for the evaluation of the sensitivity of the learning rate and the choice of the optimiser. Both experiments are evaluated based on a prequential test-then-train evaluation with a window size of $1,000$ instances, and all instances of the data stream are passed through an online standard scaling algorithm included from the *river* library. We apply for both experiments identical neural architectures, whereby within the experiments for the learning rate, we add the implementation of a logistic regression from the *river* framework. Furthermore, to depict eventual differences between the implementation of the logistic regression in *river* and our wrapper class, we implement a logistic regression architecture according to Listing 8.1. To measure the influence of different learning rates and optimisers, we implement a static *MLP* architecture that contains two hidden layers with five neurons, each followed by a Softmax layer. The Dynamic *MLP* architecture implements its architecture according to the number of input features, where the first hidden layer contains five times the number of features neurons and the second layer two times the number of features. The last layer is, similar to the static *MLP* architecture, a Softmax layer.

For the evaluation of the impact of the learning rate, we choose learning rates ranging from 10^{-9} to 100. This range origins from commonly applied learning rates in batch learning that range between 0.1 to 10^{-5} [101] that we enlarged for our evaluation. We base this experiment on a traditional *SGD* optimiser since it establishes the weight adaption solely on the learning rate and is not influenced by other parameters such as the momentum.

The evaluation for the choice of the underlying optimiser is performed based on the optimisers *SGD* and *Adam* presented within the foundations in Section 2.2.4.5. The iterative processing of instances depicted in Algorithm 6, as well as the nature of data streams regarding time dependencies, may favour optimisers with decay factors or momentum. The *SGDHD* optimiser was introduced by Rumelhart et al. [201] in 1986 and adds a momentum to *SGD* that remembers the weight update at each iteration. It determines the next weight update as a linear combination of the gradient and the previous update. This momentum might be beneficial in *online learning*, especially for the adaptation to concept drifts. As *Adam* is an extension to *RMSProp*, *AdamW* is an extension to *Adam*, proposed by Loshchilov and Hutter [158] where the L_2 regularisation factor of *Adam* is decoupled from the optimisation process and thus improves the generalization performance. The experiments with different underlying optimisers are performed based on the results of the experiments with a variable learning rate and thus set to 0.01.

8.2.3 Suitability of Neural Networks

The experiments towards the general suitability of *NN*s are based on the results of the *HPO* experiments. We evaluate in this section the suitability of *NN*s on data streams, that are similar configured to the data streams depicted in Section 8.2.2 Further, we base the evaluation on an evolving accuracy metric. To show the general suitability of *NN*s in *online learning*, we present in Table 8.3 the configuration for this experiment.

Table 8.3: Experimental Setup suitability of *NN*

Parameter	Setting
Stream Generators	Agrawal$(50,000)$, Agrawal(50), Covertype, Electricity, HYP$(50, 0.0001)$, HYP$(50, 0.001)$, LED, RBF$(10, 0.0001)$, RBF$(10, 0.001)$, *SEA*(50000), *SEA*(50), Sine
Metric	Accuracy
	Memory consumption in Mb
	Time efficiency
Preprocessing	Standard Scaling
Models	
River Logistic Regression	learning rate: 0.01
	optimiser: *SGD*
Hoeffding Tree	split criterion: information gain
	max depth: no
	max size 100Mb
Gaussian Naive Bayes	
Torch Logistic Regression	learning rate: 0.01
	optimiser: *SGDHD*
Torch Static *MLP*	learning rate: 0.01
	optimiser: *SGDHD*
	architecture: 2 hidden layers, 5 neurons each final Softmax layer
Torch Dynamic *MLP*	learning rate: 0.01
	optimiser: *SGDHD*
	architecture: $5\times$ #features, $2\times$ #features, final Softmax layer

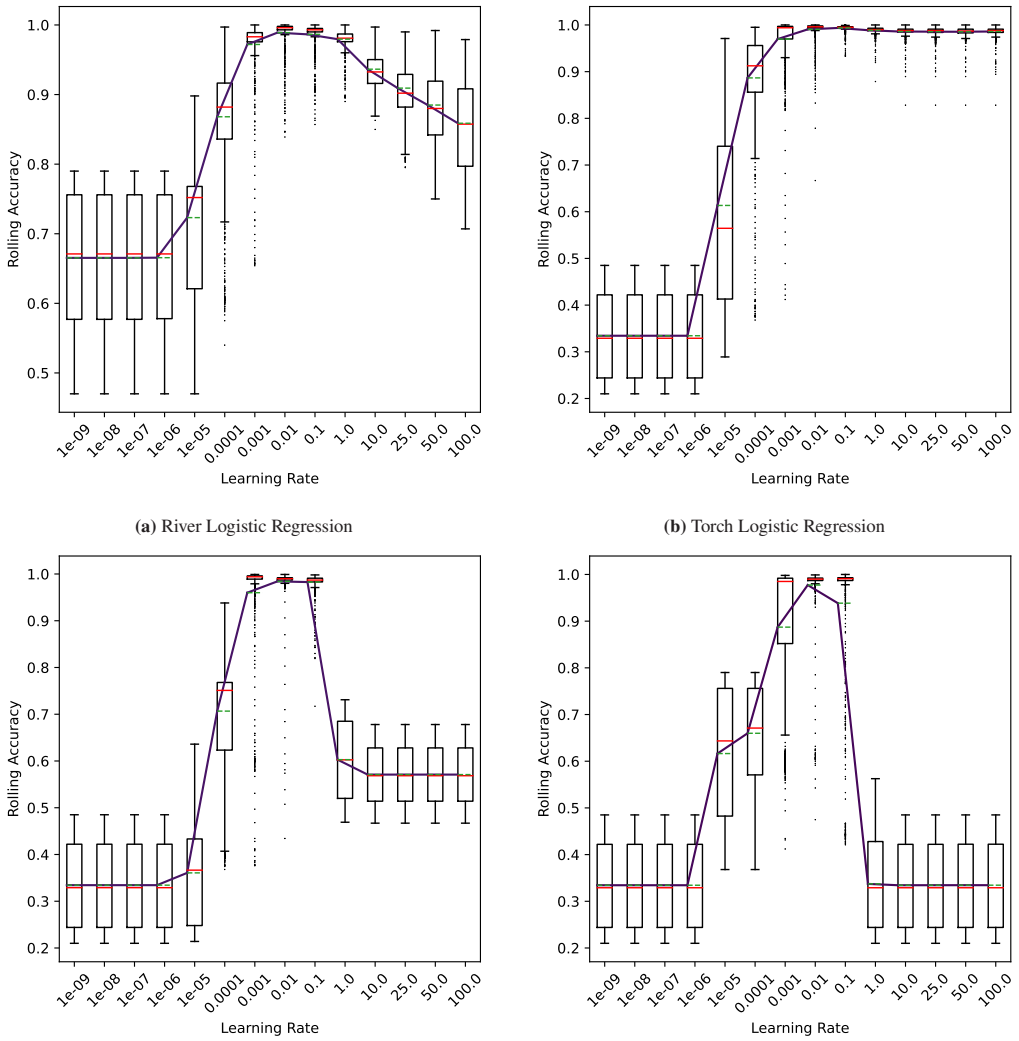
We consider in this experiment different algorithms from the *river* library to evaluate the competitiveness of different neural architectures. Additionally to the *MLP* networks implemented within the `PyTorch2RiverClassifier`, we select *GNB*, *HT* and the Logistic Regression from the *river* library. As the Logistic Regression implementation in *river* supports only a small fraction of optimisers, we chose *SGD* as the underlying optimiser. For the *NN*s we select the identical architectures as depicted in Section 8.2.2, whereby we use the results of the experiment towards a reasonable default parametrisation in that we set a learning rate of 0.01 and a *SGDHD* optimiser. Furthermore, we evaluate in this experiment not only the Rolling Accuracy but also the memory consumption in Mb and the avg. time needed to evaluate $100,000$ instances.

8.3 Results

In this section, we discuss the results achieved by the conducted experiments. The first experiment investigated the search for a suitable learning rate and optimiser for an underlying *NN*. This experiments aim is to investigate according to the *Scikit-learn* design principles the requirement *R III–5*. Within the second experiment, we show the general competitiveness of *NN*s applied on data streams and thus substantiate a possible application of *NAS*.

8.3.1 Default Parametrisation

The experiment towards a reasonable default parametrisation (*R III–5*) is two-folded. We identified within the implementation of the `PyTorch2RiverClassifier` class, besides the neural architecture, two parameters that need to be set; (i) the learning rate and the (i) underlying optimiser. In Figure 8.2, we depict the box-plots of the measured rolling accuracies over the evolving *SEA*(50) data stream of the different learning rates and models. We depict a detailed view of the results for this experiment within the Appendix in Table A.3 for the Logistic Regression as smallest *NN* and in Table A.4 for the static and dynamic *MLP* classifier.



(a) River Logistic Regression

(b) Torch Logistic Regression

(c) Torch Static *MLP* Classifier

(d) Torch Dynamic *MLP* Classifier

Figure 8.2: Impact of Learning Rate for *SEA*(50) Concept Drift Stream

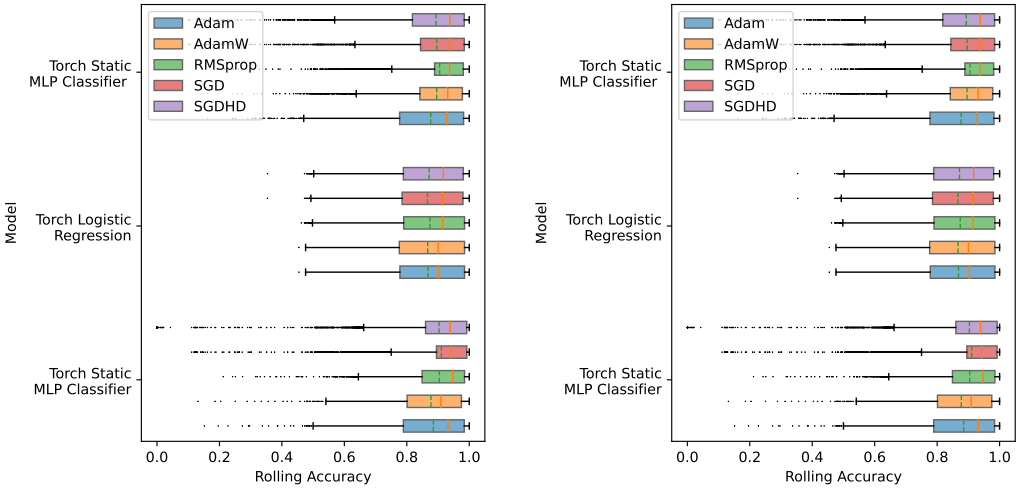
Figure 8.2 shows exemplarily on the *SEA* data stream that lower learning rates perform worse than higher learning rates. The results furthermore show that the average performance of the *NN* increases from a learning rate of 10^{-5} for all classifiers. But also, the variance decreases significantly as the learning rate increases. For the Logistic Regression models, the peak performance is achieved with a learning rate of 0.01. However, it is noticeable that the performance of the Logistic Regression model implemented in *river* drops stronger from a learning rate of 0.1 than that of the *PyTorch* implementation. The *MLP* classifiers act similarly to the Logistic Regression models. The peak performance is achieved at a learning rate of 0.01, whereby again, besides the performance, the variance of the rolling accuracy decreases significantly. While the static *MLP* model starts to decrease in performance and increase in variance at a learning rate of 0.1 to a plateau at 60%, the performance of the dynamic *MLP* classifier drops to a performance of 30%. Similar results are obtained for the other data streams in Tables A.3 and A.4. However, we assume 0.01 as a learning rate best suited as the default parameter as the average rolling accuracy is the highest. The low variance, however, indicates a better adaptation to concept drifts.

Table 8.4 shows the results regarding the choice of a suitable optimizer. In Figure 8.3, we depict further the learning curves for the *SEA*(50) data stream. As for the results for the experiments towards a suitable learning rate, depict according to Table 8.4, the results for the *PyTorch* Logistic Regression in Table A.1 and for the static *MLP* classifier in Table A.2 within the appendix.

Table 8.4: Rolling Accuracy comparison Torch Dynamic *MLP* classifier considering the optimisers *Adam*, *AdamW*, *SGD*, *SGDHD* and *RMSprop* Accuracy is measured as the average rolling percentage of examples correctly classified. The best model accuracies are indicated in boldface

Data Stream	**Torch Dynamic *MLP* Classifier**				
	Adam	**AdamW**	**RMSprop**	**SGD**	**SGDHD**
Agrawal(50,000)	**76.28 ±0.05**	75.5 ±0.05	75.47 ±0.06	75.25 ±0.07	75.3 ±0.07
Agrawal(50)	91.5 ±0.1	**91.6 ±0.05**	90.75 ±0.08	88.98 ±0.12	85.32 ±0.11
Covertype	80.15 ±0.11	78.04 ±0.1	**91.34 ±0.1**	91.27 ±0.09	85.73 ±0.22
Electricity	81.54 ±0.06	79.37 ±0.08	**90.85 ±0.03**	86.72 ±0.07	85.37 ±0.08
HYP(50, 0.0001)	78.71 ±0.04	81.89 ±0.04	84.09 ±0.05	92.02 ±0.06	**92.22 ±0.06**
HYP(50,0.001)	80.06 ±0.04	81.22 ±0.04	85.19 ±0.04	91.4 ±0.06	**91.52 ±0.06**
LED	74.37 ±0.05	72.48 ±0.05	73.48 ±0.04	74.26 ±0.1	**74.53 ±0.1**
RBF(10,0.0001)	**99.52 ±0.01**	99.31 ±0.01	99.5 ±0.02	99.37 ±0.05	99.38 ±0.05
RBF(10,0.001)	97.99 ±0.01	97.86 ±0.02	98.52 ±0.02	**98.93 ±0.05**	98.89 ±0.05
SEA(50)	**98.65 ±0.01**	97.39 ±0.01	98.52 ±0.01	97.73 ±0.06	97.77 ±0.06
SEA(50000)	**93.39 ±0.02**	92.6 ±0.02	93.09 ±0.02	91.65 ±0.06	91.7 ±0.06
Sine	98.59 ±0.02	97.56 ±0.02	**98.7 ±0.01**	95.7 ±0.12	95.75 ±0.12
Avg. Accuracy	87.56	87.07	89.96	**90.27**	89.98
Avg Rank	2.67	3.92	2.53	3.25	**2.33**

Table 8.4 shows, proximate results over changing optimisers. While the choice for the learning rate influenced the model's rolling accuracy in the order of magnitude of up to 70%, the choice of the underlying optimiser influences the rolling accuracy in the order of magnitude of $\sim 5\%$. In addition to the the average rolling accuracy for each data set, we depict in Table 8.4 the averaged rolling accuracy over all data streams and the average rank achieved by the underlying optimiser. These results show, that *SGD* achieves for the dynamic *MLP* classifier in average better accuracy scores, however, *SGDHD* achieves the best average rank. In case of the other classifiers it turns out, that *SGDHD* performance marginally better than the other optimisers.



(a) *SEA*(50) **(b)** *SEA*(50000)

Figure 8.3: Impact of the underlying optimiser on *SEA*(50) and *SEA*(50000) evaluated on 100, 000 instances

In Figure 8.3, we illustrate the impact of the underlying optimiser on *SEA*(50) and *SEA*(50000). As depicted in Section 8.2.2, the *SEA* stream contains three drifts that are applied within a width of $w = 50$ (abrupt drift) and $w = 50,000$ (continuous drift) Comparing both figures shows that the underlying optimiser performs equally well in both change widths. Furthermore, it shows that the optimiser only marginally influences the performance of the Logistic Regression model. The impact of the optimiser for the static as well as the dynamic *MLP* is higher in that *RMSProp* shows, compared to the other optimisers, a comparably lower variance. However, taking the results in of the other architectures in Table A.1 and A.2 into account it turns out that *SGDHD* is a suitable optimiser for *NN*s in *online learning*. By evaluating the models based on a rolling accuracy, we furthermore showed that *NN* are capable of adapting to temporal changes (*R I–5*) In the following, we base our evaluations on *SGDHD*.

Concluding the results for a default parametrisation of *NN*, we showed in this section that a learning rate of 0.01 and *SGDHD* as underlying optimisers are suitable default configurations. Furthermore, it emerges that the *DL* models (dynamic and static *MLP*) achieve a higher average rolling accuracy.

8.3.2 Suitability

In this section we evaluate the general suitability of *NN* and compare according to the experimental setup depicted in Section 8.2.3 an evaluation considering various models from the *river* framework, as well as the requirements for *online learning*. While *NN* follow the *online learning* requirements *R I–1*, *R I–4*] by nature and the adaptivity of *NN* can be derived from the experiments based on the evaluations towards a reasonable default parametrisation, we investigate in this evaluation the requirements *R I–2* and *R I–3*. In Figure 8.4, we present a resource aware evaluation of the *SEA*(50) data stream.

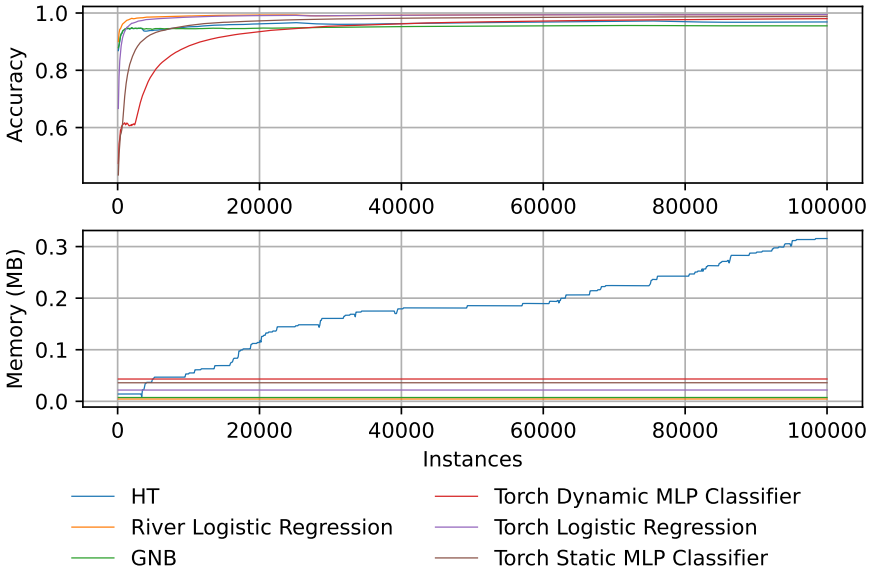


Figure 8.4: Accuracy curve and memory (in Mb) for various *NN*s and *river* algorithms

Looking at accuracy, we see that *NN* models show competitive performance compared to *HT*, *GNB* and the Logistic Regression model implemented in *river*. However, the dynamic *MLP* model adapts comparably slower to the underlying distribution, but it emerges that all models are resistant to the concept drifts implemented within *SEA*. Denote that we evaluate this experiment based on an accuracy metric that gets

insensitive to performance changes over time. We choose in this experiment an accuracy metric since the emphasis lies on the model's performance rather than on its adaptation. Furthermore, we depict in Figure 8.4 the memory consumption of each algorithm. Overall, the results indicate that the *HT* model increases its memory consumption in comparison to the other algorithms. This brings additional advantage of *NN* in *online learning*. While the *HT* builds a decision tree that increases over time, the neural architecture when applying *NN* remains fixed, and thus the memory consumption does not increase over time. To solve the issue of *HT*'s memory consumption, a *HAT* can be applied, or a maximum depth of the tree can be set.

Table 8.5: Accuracy comparison of *EvoAutoML* against baselines. Accuracy is measured as the final percentage of examples correctly classified. The best individual accuracies are indicated in boldface

Data Stream	GNB	HT	River Logistic Regression	Dynamic MLP Classifier	Torch Logistic Regression	Static MLP Classifier
Agrawal(50,000)	60.37 ±0.06	77.06 ±0.02	59.87 ±0.02	**75.33 ±0.04**	59.03 ±0.02	74.98 ±0.03
Agrawal(50)	62.51 ±0.09	**90.55 ±0.03**	63.90 ±0.03	89.10 ±0.08	63.44 ±0.03	80.71 ±0.04
Covertype	37.35 ±0.07	79.39 ±0.11	22.03 ±0.04	**91.42 ±0.08**	87.90 ±0.05	87.25 ±0.11
Electricity	76.14 ±0.03	79.78 ±0.03	83.70 ±0.01	87.03 ±0.06	85.51 ±0.01	**87.11 ±0.06**
HYP(50, 0.0001)	93.38 ±0.02	83.95 ±0.01	**92.81 ±0.02**	92.20 ±0.07	91.57 ±0.02	92.51 ±0.07
HYP(50,0.001)	85.29 ±0.02	82.40 ±0.01	**92.17 ±0.02**	91.59 ±0.06	91.37 ±0.01	91.94 ±0.07
LED	76.60 ±0.02	63.62 ±0.07	13.91 ±0.01	74.56 ±0.12	**76.63 ±0.02**	72.86 ±0.11
RBF(10,0.0001)	94.34 ±0.02	88.18 ±0.03	30.35 ±0	99.64 ±0.06	**99.77 ±0**	96.03 ±0.08
RBF(10,0.001)	63.60 ±0.11	71.81 ±0.07	29.99 ±0	**99.19 ±0.05**	97.33 ±0.01	97.51 ±0.08
RBF(50,0.0001)	67.74 ±0.08	84.13 ±0.02	41.17 ±0	**98.49 ±0.1**	96.64 ±0.02	94.79 ±0.12
RBF(50,0.001)	35.41 ±0.13	58.38 ±0.06	36.69 ±0.01	**96.12 ±0.09**	83.71 ±0.02	74.64 ±0.09
SEA(50,000)	94.72 ±0.01	95.67 ±0.01	96.78 ±0	95.53 ±0.07	**96.79 ±0.01**	96.34 ±0.04
SEA(50)	95.52 ±0.01	96.88 ±0.01	99.41 ±0.01	98.03 ±0.07	**99.44 ±0.02**	98.78 ±0.05
Sine	93.83 ±0.01	98.77 ±0.01	98.39 ±0.01	95.90 ±0.12	98.40 ±0.01	**98.78 ±0.03**
Avg. Accuracy	74.06	82.18	61.51	**91.72**	87.68	88.87
Avg. Rank	4.79	4.00	4.21	**2.57**	2.79	2.64

In Table 8.5, we depict the achieved results on all data streams. It shows that the dynamic *mlp* classifier achieved the best-averaged accuracy with 91.72% and the best rank $(2, 57)$. Furthermore, the results indicate that *NN*s implemented within the `PyTorch2RiverClassifier` perform better on the selected data steams. Looking at each individual data stream, the dynamic *MLP* model dominates with 91.42% on Covertype, and the static *MLP* dominates with 87.11% on the Electricity data stream against the other models. Overall, the results indicate that *NN* are competitive to other models in their predictive performance. To evaluate the time (*R I–2*) and memory (*R I–2*) consumption of *NN*s, we depict in Table 8.6 the memory consumption, the average time needed to iterate over the data streams and the RAM-hours (in Mb hours) employed during iteration.

Table 8.6: Comparisons of memory consumption (in Mb) and Avg. Time (in s) of *NN*s against different baselines. One RAM-Hour equals to 1 Mb of RAM deployed for 1 hour.

Data Stream	GNB	HT	River Logistic Regression	Dynamic MLP Classifier	Torch Logistic Regression	Static MLP Classifier
Agrawal(50,000)	0.016	0.933	**0.006**	0.045	0.023	0.037
Agrawal(50)	0.016	0.920	**0.006**	0.045	0.023	0.037
Covertype	0.263	1.750	**0.018**	0.057	0.036	0.050
Electricity	0.015	0.466	**0.006**	0.044	0.023	0.037
Hyperplane(50, 0.0001)	0.078	3.710	**0.015**	0.054	0.032	0.046
Hyperplane(50,0.001)	0.078	3.710	**0.015**	0.054	0.032	0.046
LED	0.051	0.346	**0.005**	0.045	0.023	0.037
RBF(10,0.0001)	0.145	2.740	**0.016**	0.055	0.033	0.047
RBF(10,0.001)	0.145	6.210	**0.016**	0.055	0.033	0.047
RBF(50,0.0001)	0.178	3.730	**0.016**	0.055	0.033	0.047
RBF(50,0.001)	0.178	2.390	**0.016**	0.055	0.033	0.047
SEA(50,000)	0.007	0.268	**0.005**	0.043	0.022	0.036
SEA(50)	0.007	0.316	**0.005**	0.043	0.022	0.036
Sine	0.006	0.210	**0.004**	0.043	0.022	0.036
Avg. Time	168.729	217.456	**116.585**	360.329	161.667	191.442
RAM-Hours	0.109	1.614	**0.007**	0.078	0.020	0.034

Considering the time efficiency (*R I–2*) of the underlying models, Table 8.6 shows that the Logistic Regression implemented in *river* consumes the least time, followed by the Logistic Regression implemented within the `PyTorch2RiverClassifier`. This behaviour is expected as the model wraps the data stream and passes it through a *PyTorch* model. It is rather remarkable that the *NN*s is capable of processing the data stream at the same magnitude of speed as the *HT* classifier. Another expected result in Table 8.6 is that the time required increases with the size of the *NN* as the number of operations increases. However, comparing the time consumption with regard to the fulfilment of *R I–2*) it shows that *NN* are more complex in the computations and thus need more computation time. This increased time consumption, however, is decent, which makes smaller *NN*s quite suitable for data streams.

Considering the memory efficiency (*R I–3*) of the underlying models, it again shows that the Logistic Regression implemented in *river* employs with 0.007 Mb hours the least RAM-hours. Followed by the Logistic Regression implemented within the `PyTorch2RiverClassifier` with 0.020 Mb hours. Accordingly to Figure 8.4, the dynamic and static *MLP* classifier consume less memory than *GNB* or *HT*. The *HT* classifier, in particular, shows its disadvantages when it is not restricted in its tree depth.

To conclude the suitability of *NN* regarding *R I–2* and *R I–3*, this section showed, that *NN*s are suitable models for *online learning*. Due to their versatile possibilities in their underlying architectures, they can be adapted to the processing time and memory requirements and are thus well suited for data streams.

8.4 Summary

This chapter presented and evaluated an *online DL* framework that wraps the popular *PyTorch DL* library into a *river* `Predictor` in case of a supervised learning task. This framework aims to simplify the development of *DL* models specially designed for data streams. This is achieved by following the *Scikit-learn* design principles that we think are an enabler towards the application of *NN* in *online learning*. As we rely in this thesis on supervised learning (preliminary classification) tasks, we emphasise the training and prediction of *NN*s in a supervised *online learning* manner. In the second part of this chapter, we focused on the fulfilment of the requirements in *online learning* and presented a resource-aware evaluation of the *NN*s in general. We further show in an empirical evaluation that incorporates the smallest *NN*, the Logistic Regression, as well as *MLP*s that the general application of (architectural smaller) *NN*s in *online learning* can be beneficial. Since

we restrict the evaluation to simple neural architectures that showed the competitiveness in *online learning* and concerning the versatile development of neural architectures in *offline learning*, a fruitful direction towards the adaptation of *NN* in *online learning* is shown.

9 Incremental HPO

This chapter investigates the automated configuration of *ML* pipelines and *NN*s on data streams. Within Chapter 7, we presented Hypothesis II and depicted the overall framework design towards the adaption of *AutoML* and *NAS*. Chapter 8 lays the foundation for online *NAS* by providing a framework for online application of *NN*s. We propose in this chapter an *AutoML* framework (Contribution III) that enables a continuous adaption to data streams by as well following the *river API* and the *Scikit-learn* design principles. Following the *river API* design within an incremental *HPO* enables in combination with the proposed *online DL* framework the application of *NAS* in an *online learning* environment. Thus, derived from Hypothesis II, we aim to answer in this chapter the following research questions.

RQ III.2 How can the hyperparameters of *ML* pipelines (*AutoML*) and *NN*s (*NAS*) incrementally be adapted to data streams?

RQ III.3 Does an incremental adaptation of hyperparameters in *AutoML* or *NAS* systems enable better performances on data streams?

In the research question (RQ III.2), we propose a general approach that enables the adaptation of hyperparameters concerning the automated configuration of *ML* pipelines and neural architectures on the data streams. Following the *CASH* definition from Feurer et al. [81] for batch learning, we formalise in Section 9.1 an *online CASH* problem. To solve the *online CASH* problem, we propose in Section 9.2 an *ensemble* approach that aims to solve the *online CASH* problem that can be applied on *NAS*. In Section 9.3, we evaluate our approach towards the search for suitable *ML* pipelines. And in Section 9.4, we apply the incremental *HPO* approach to a *NAS* search space and evaluate it with respect to the *online learning* requirements (I). Parts of this chapter have been published in "Evolution-Based Online Automated Machine Learning" [141] by Cedric Kulbach, Jacob Montiel, Maroua Bahri and Albert Bifet on *PAKDD 2022*.

9.1 Problem Formalisation

As part of the related work we presented in Section 3.3 *online* ensemble techniques that aim to to adapt to data streams by training a set of homogeneous (e.g. *LB* [24], *OB* [181] , *SRP* [99]) or heterogeneous (e.g. *BLAST* [233]) model configurations. However, the configuration of heterogeneous models during the evolving data stream has not been part of the research. The configuration of models during the stream entails multiple difficulties. The underlying models are mostly configured at the beginning of the data stream, and a reconfiguration leads to a new model that needs to adapt to the data stream. Furthermore, the configuration of *ML* pipelines accordingly to the *offline CASH* problem that concatenate different algorithms in *online learning* has yet not been part of the research. Thus, the question of changing and adapting the configuration of an algorithm as well as the orchestration of models without infringing the requirements [23] for online learning remains open. Following the definition from [253], a *ML* pipeline structure $g \in G$ can be modelled

as an arbitrary *DAG*, where each node represents an algorithm $A \in \mathcal{A}$. The *online CASH* problem is defined in Definition 20.

➢ **Definition 20.** Online *CASH*, adapted from [81]
Let $\mathcal{A} = \{A^{(1)}, \ldots, A^{(R)}\}$ be a set of step independent algorithms, and let the hyperparameters of each algorithm $A^{(j)}$ have a domain $\Lambda^{(j)}$. Further, let $\mathcal{S} = e_1, e_2, \ldots, e_t, \ldots$ be an ordered sequence of examples of possibly infinite length and let t be the current observed example. Further, let $S^- = e_0, \ldots, e_t$ be an ordered sequence of past examples. Each example $e_i = \{x_i, y_i\}$ is a tuple of p predictive attributes $x_i = (x_{i,1}, \ldots, x_{i,p})$ and the corresponding label y_i. Let $\mathcal{L}(\mathcal{P}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}(\mathcal{S}^T), \mathcal{S}^V)$ denote the loss that algorithm combination $P^{(j)}$ achieves on a subset of validation examples $\mathcal{S}^V \subset \mathcal{S}^-$ when trained on $\mathcal{S}^T \subset \mathcal{S}^-$ with hyperparameters $\overrightarrow{\lambda}$. Denote that $\mathcal{S}^T \cap \mathcal{S}^V = \emptyset$.

Then the Online *CASH* problem is to find the joint algorithm combination and hyperparameter setting that minimises based on a validation protocol $\mathcal{V}$ the loss:

$$g^*, \overrightarrow{A}^*, \overrightarrow{\lambda}^* \in \underset{P^{(j)} \in \mathcal{P}, \lambda \in \Lambda^{(j)}, A \in \mathcal{A}, g \in G}{\arg\min} \mathcal{V}(\mathcal{L}, \mathcal{P}_{g, \overrightarrow{A}, \overrightarrow{\lambda}}, \mathcal{S}^T, \mathcal{S}^V) \tag{9.1}$$

The changes to the *offline CASH* problem in comparison to Definition 20 do not appear to be major at first glance. Whereby the nature of data streams poses new challenges in that the order of the data instances is crucial. This ordering entails a new dimension to the *CASH* problem in that the question arises when to adapt to the configuration of the underlying model. The change from data sets to data streams, thus, excludes k-fold cross-validation (see Section 2.6) and includes evaluation protocols suitable for *online learning*, such as *test-than-train* or *prequential* evaluation. In ensemble learning (e.g. *ARF* [98] or *HAT* [21]) or in conceptual approaches towards the application of *offline AutoML* approaches on data streams such as proposed by Imbrea [121] concept drift detectors are used to determine when to change the underlying configuration. The formalisation in Definition 20 reveals two major tasks towards the application of *AutoML* to data streams; The determination of (i) when to adapt the underlying *ML* pipeline or neural architecture as well as (ii) a variable concatenation of algorithms in the context of *AutoML*. In the following, we propose our approach that aims to solve the defined *online CASH* problem.

9.2 Approach

Inspired by the *CASH* problem, a *GA* approach and *online ensemble* techniques *OB* [181, 180] we propose in this section an approach that enables online training in a high-dimensional algorithm- and hyperparameter-search spaces. However, the *CASH* solution does not consider the adaption of parameters in an evolving data stream environment so far; on the other hand, the established online ensemble algorithms are only capable of processing a small set of homogeneous algorithms. Whence, our proposal uses a *GA* approach, which naturally adapts its configurations within a small ensemble (population) to enable the adaption of a large algorithm- and hyperparameter-search space to evolving data streams. The core of our training and adaption procedure depicted in Algorithm 8 is a *GA* inspired by Real et al. [197]. Accordingly, to the underlying *GA* approach, we refer to our approach as *EvoAutoML* (*Evolution-based Online Automated Machine Learning*).

The algorithm takes a data stream $\mathcal{S}$, a population size p, a sampling rate f_{SS} as well as a loss function $\mathcal{L}$ and a configuration space $\mathcal{A}, \Lambda, G$. When considering a data stream of a fixed length, the number of cycles

of the underlying *GA* can be referred to as $\lfloor t/f_{SS} \rfloor$. However, due to the unbound nature of data streams, the number of cycles is infinite, and thus the *GA* evolves as the data stream does. Whereby the sampling rate f_{SS} estimates how many instances are processed without evolution and mutation steps. Following the *AutoML* system design in *offline learning*, the loss function $\mathcal{L}$ is externally set. Regarding *NAS*, we assume furthermore that the search space that contains all possible configurations $g \in G$, algorithms $A \in \mathcal{A}$ and algorithms configurations Λ is set externally and additionally required. In Algorithm 8, we present the training procedure of *EvoAutoML*.

Algorithm 8 EvoAutoML Training

1: **Input:**
2: Data stream $\mathcal{S}$, population size p, sampling rate f_{SS}, loss function $\mathcal{L}$, configuration space $\mathcal{A}, \Lambda, G$
3: **Output:**
4: Set of suited algorithms configurations:
5: $p^* = \{\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(P)}\}$
6:
7: $P \leftarrow \emptyset$ ▷ Initialization
8: **while** $|P| < p$ **do**
9: $\mathcal{P} \leftarrow$ Random$(G, \mathcal{A}, \Lambda)$
10: $P \leftarrow P \cup \mathcal{P}$
11: **end while**
12: $t \leftarrow 0$
13: **if** e_t **then** ▷ Start Data Stream
14: **if** $t \mod f_{SS} == 0$ **then**
15: $\mathcal{P}^{best} \leftarrow \min_{\mathcal{P} \in P} \mathcal{L}^{(\mathcal{P})}(\mathcal{P}(\mathcal{S}^T), \mathcal{S}^V)$
16: $\mathcal{P}^{weak} \leftarrow \max_{\mathcal{P} \in P} \mathcal{L}^{(\mathcal{P})}(\mathcal{P}(\mathcal{S}^T), \mathcal{S}^V)$
17: $\mathcal{P}^{mut} \leftarrow$ Mutate$(\mathcal{P}^{best})$
18: $P \leftarrow P \cup \mathcal{P}^{mut}$
19: $P \leftarrow P \backslash \mathcal{P}^{weak}$
20: **end if**
21: $\omega \leftarrow Poisson(6)$
22: **for** $\mathcal{P} \in p$ **do** ▷ Update Population
23: **loop** ω
24: $\mathcal{P}$.fit(e_t)
25: **end loop**
26: **end for**
27: $t \leftarrow t + 1$
28: **end if**

In Algorithm 8, we refer to *ML* pipelines $\mathcal{P}$ that are configured by g, $\overrightarrow{A}$ and $\overrightarrow{\lambda}$. This notation can be adapted or replaced to the search space of neural architectures, where a *NN* f is configured by g, $\overrightarrow{z}$ and $\overrightarrow{\lambda}$.

The population of the underlying *GA* in Algorithm 8 is initialised within lines 7 - 12, where configurations are picked randomly from the search space and appended to a population set P. Notice that by initialising P with random online learning pipelines before the data stream starts, the algorithm is able to predict at any time (*R I–4*). The data stream and thus the evolutionary steps of the *GA* start in line 13 of Algorithm 8. The evolution steps are applied with a sampling rate of f_{SS} (line 14). The mutation incorporates a mutation step that is applied based on the best pipeline configuration within P. The mutated pipeline $\mathcal{P}^{best}$ replaces the weakest pipeline configuration. The applied mutation step follows the implementation of Real et al. [197] where one parameter of the search space is randomly changed. Since the metric $\mathcal{L}$ evolves with the data stream and the model's performance, each pipeline configuration in P requires a separate metric function

$\mathcal{L}^{(\mathcal{P})}$. To follow the test-than-train protocol, the population is trained after the mutation step based on the new instance e_t in lines 21-25. Accordingly to Oza and Russell [181], we repeat the training on the instance e_t following a $Poisson$ distribution. Whereby the number of trainings is not separately chosen for each model in P and for the expected value of the $Poisson$ distribution we follow Bifet et al. [24] with a $Poisson(6)$ distribution.

Considering the *online learning* requirements, our approach follows ***R I–4*** by initialising a set of *online learning* algorithms and further processing one instance of the stream at a time ***R I–1***. Our approach aims to continuously adapting an ensemble of algorithms to the data stream (***R I–5***) accordingly to Oza and Russell [181] and Bifet et al. [24], but also considers the underlying configurations and heterogeneous models. However, the adaptation towards the memory and time consumption of *EvoAutoML* is shown within the results for *AutoML* in Section 9.3 and *NAS* in Section 9.4. *EvoAutoML* trains a population of *ML* pipelines, or *NN*s and is thus an ensemble learner that has access to a population of trained pipelines at each point of the data stream. This leads to a further advantage that *EvoAutoML* predicts an unlabelled instance based on a majority voting. During the evaluation and development of Algorithm 8 it emerged that a hard majority voting performed best on the evaluated data streams. A hard majority voting predicts for a classification task an instance $\hat{y}$ based on

$$\hat{y}_i = mode\{\mathcal{P}.predict(e_t) \in P\} \tag{9.2}$$

where the class with the most votes is elected as prediction value. Under the assumption that the configuration of the underlying *ML* pipeline or *NN* evolves accordingly to the never ending data stream, we approach a *GA* and presented within Algorithm 8 an automated configuration strategy towards stream adaptation. Further, we implemented the framework likewise the `PyTorch2RiverClassifier` depicted in Chapter 8, *river* `Estimator` that implements Algorithm 8 within the `learn_one` method and the majority voting within the `predict_one` method. The underlying configuration space $(G, \mathcal{A}, \Lambda)$ is defined within the initialisation of the *EvoAutoML* estimator, whereby the graph structure $g \in G$ and the algorithm space are combined within a Python dictionary. The parametrisation space Λ is as well a configurable Python dictionary, whereby possible configuration candidates are represented within iterables. In the following, we evaluate *EvoAutoML* on *ML* pipelines.

9.3 Online AutoML

In this section, we evaluate the approach depicted in Section 9.2 based on *ML* pipelines. Besides the predictive performance and thus the capabilities in adaptation, we investigate within the evaluation the time (***R I–2***) and memory (***R I–3***) requirements. In Section 9.3.1, we depict the experimental setup that incorporates the used data streams as well as the search space to build *ML* pipelines. Based on the experimental setup, we present the achieved results in Section 9.3.2.

9.3.1 Experimental Setup

This section provides the experimental setup for the evaluation of *EvoAutoML*. As in Chapter 8, we apply a *test-then-train* evaluation. In addition to the fulfilment of the requirements ***R I–4*** and ***R I–1***, we evaluate our approach against state-of-the-art single and ensemble algorithms. We show that *EvoAutoML* is compatible with recent online algorithms and thus fulfils for *AutoML* all requirements of [23] (*R* I). In Table 9.1, we present an overview of the experimental setup to evaluate the predictive performance and as well the time and memory consumption of *EvoAutoML*. We executed all experiments on an *Intel(R) Xeon(R) Platinum*

8180M CPU with 2.50 GHz base clock and 1.5 terabytes of RAM without consideration of further *GPU* acceleration.

Table 9.1: Experimental Setup *EvoAutoML*

Parameter	Setting
Stream Generators	Agrawal(50, 000), Agrawal(50), Covertype, Electricity, HYP(50, 0.0001), HYP(50, 0.001), LED, RBF(10, 0.0001), RBF(10, 0.001), RBF(50, 0.001), RBF(10, 0.0001), *SEA*(50), Sine
Metric	Accuracy
	Memory consumption in Mb
	Time efficiency
Preprocessing	Standard Scaling (except for *AutoML*)
Models	
ARF	#models: 10
	max memory: 32 Mb
OB	#models: 10
	base model: *HT*
GNB	-
k-NN	#neighbors: 5
	window_size: 1000
HT	max memory: 32 Mb
EvoAutoML	population: 10
	f_{SS}: 1000
	Preprocessing: Standard Scaling, Missing Value Cleaner, Min-Max Scaler
	Models: *GNB*, *HT*, *k-NN*, *Logistic Regression*

According to Chapter 8, we evaluated *EvoAutoML* on data stream generators as well as real-world data sets that are iteratively passed to the underlying algorithms. We sampled within the generators $1M$ instances, considering all available instances for the real-world data streams. Further, we evaluate *EvoAutoML* against state-of-the-art single as well as ensemble learners. The reference models use standard scaling to preprocess the data stream and are executed on their default configuration. The *ARF* [98] classifier uses ten random forest models and a *ADWIN* change detector to adapt the underlying ensemble to concept drifts. Further, we limit the memory consumption to 32 Mb. The *OB* [181] model builds its ensemble based on ten *HT* algorithms, that are limited in their memory consumption to 32 Mb. The *OB* ensemble has thus a maximal memory consumption of 320 Mb. For the evaluation against single models, we choose *GNB*, *k-NN* and *HT* as reference models, that are also part of the *EvoAutoML* models step.

The configuration of *EvoAutoML* can be depicted by the configurations space $(G, \mathcal{A}, \Lambda)$, whereby it uses two algorithm types that can be categorised into (i) *preprocessors* $A^{(i)}$ and (ii) *predictors* $A^{(ii)}$, and can be variably linked with each other. The preprocessing step $A^{(i)}$ can either be a *Missing Value Cleaner*, *Min-Max Scaler*, or a *Standard Scaler* ($|A^{(i)}| = 3$). Within the prediction step *EvoAutoML* can configure four different algorithms, namely *GNB*, *HT*, *k-NN*, and *Logistic Regression*, that have been chosen for their different modes of operation. Thus in total *EvoAutoML* is capable of configuring $3 \times 4 = 12$ different algorithm pipelines.

However, all algorithms $Ain\mathcal{A}$ can furthermore be parametrised by their domain Λ. While the preprocessors $A^{(i)}$ do not differ in the parametrisation, the predictors $A^{(ii)}$ can be parametrised according to their underlying modes of operation. For example, the *k-NN* model varies by the number of neighbours and its window size or the *HT* model by its maximal depth or on which basis the underlying tree splits should be applied. In total the configuration space of *EvoAutoML* considers of 174 possible *ML* pipeline, whereby the pipeline structure G is fixed by $A^{(i)}$ followed by $A^{(ii)}$.

Considering this configuration space shows already the advantage of *EvoAutoML*. While the ensemble learning techniques are based on homogenous models or *ML* pipelines that do not consider different or changing configurations, *EvoAutoML* is parametrised by a configuration space and thus is capable of handling

a diverse set of pipeline configurations $\mathcal{A}, \Lambda$. Furthermore, as for *ARF* and *OB*, we set the population size to 10 *ML* pipelines that are evaluated at once and a sampling rate of $f_{SS} = 1000$. Denote that both parameters have a mutual influence on the predictive and time and memory performance. A high population number and a low sampling rate lead to a broad exploration but also to computational expensive training updates. It also leads to low exploitation, as the child pipeline configuration (see Algorithm 8) has no prospect of prevailing against the already established algorithms within the population. During the implementation and evaluation, the configuration $P = 10$, $f_{SS} = 1000$ was accordingly to Requirement *R III–5* found to be a compromise between predictive performance and the number of resources required. Regarding the computational complexity for large search spaces, we evaluated the related algorithms in their proposed configuration to pursue the question of the best performing approach.

9.3.2 Results

In this section, we present accordingly to the proposed approach in Section 9.2 and the experimental setup in Section 9.3.1 the results of *EvoAutoML* with regard to the automated configuration of *ML* pipelines. The results show that *EvoAutoML* is capable of outperforming state-of-the-art algorithms not only in their predictive performance but also in their memory and time consumption. We present, according to the evaluation of the *online DL* framework, the results against single algorithms. However, as *EvoAutoML* is a *ensemble learning* algorithm, we consider within the results ensemble learning algorithms as comparative models. Following Requirement *R I–5*, we depict in Figure 9.1 the development of the accuracy curve over the course of the Covertype data stream.

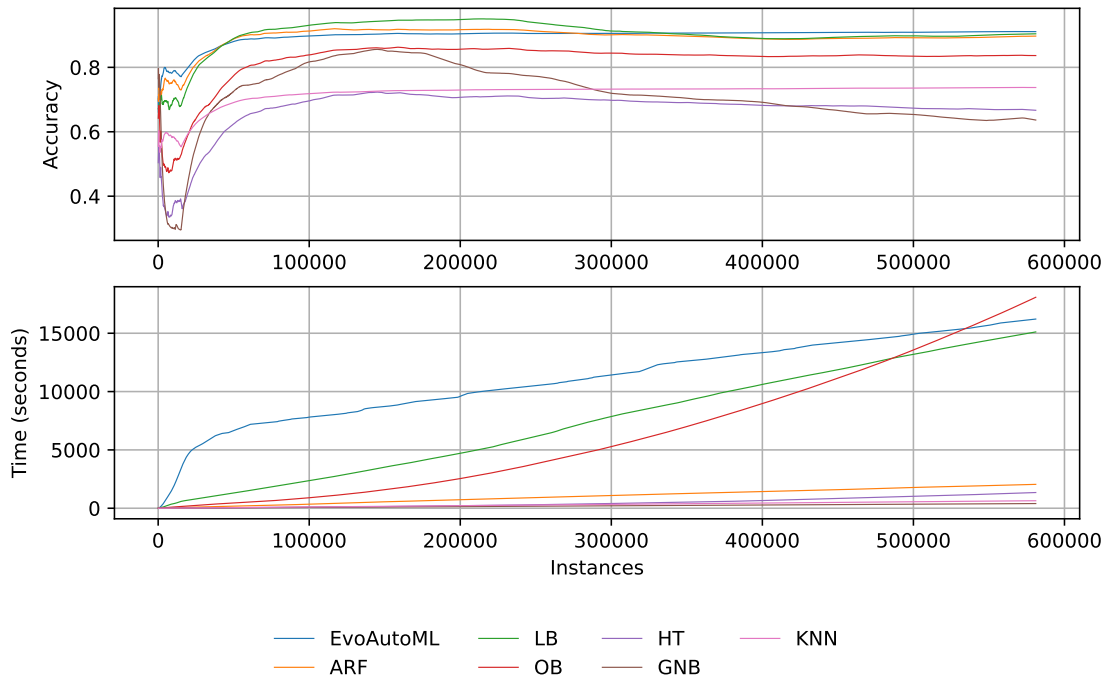


Figure 9.1: Accuracy curve and time (in seconds) for *EvoAutoML* and baseline algorithms on Covertype.

Comparing the single and *ensemble learning* algorithms, Figure 9.1 shows significant differences within the development of the accuracy curve. While both adapt within the first $\sim 50,000$ instances to the underlying data distribution and reach a performance plateau, the *ensembles* performance is significantly higher than the

performance of the single algorithms. Further, the performance of some algorithms (*GNB*, *LB*, *ARF*) drops after $\sim 225,000$ instances, which could be due to a concept drift within the Covertype data set [27]. *LB* and *ARF* achieve at the beginning of the stream the best peak performances, but decreases slightly at $\sim 225,000$ instances; however, the accuracy curve of *EvoAutoML* remains stable over the stream and achieves with 91.09% marginally better performances than *ARF* (89.70%) and *LB* (90.41%). *OB* achieves with 83.66% better performances than the single best algorithms, *HT* (66.67%), *GNB* (63.64%) and *k-NN* (73.74%), but performs worst within other *ensemble learners* on Covertype.

A significant drawback of *ensemble learners* comes apparent when considering the time needed to process the Covertype data stream. The *ensembles* need considerably more time to process incoming data instances. While *EvoAutoML* needs at the beginning of the data stream in comparison to the other models the most time, it increases as *LB* linearly within the time consumption depicted in Figure 9.1. *OB*, however, increases super-linear and considering the unbound nature of data streams, it might lead to runtime problems. Furthermore, the single algorithms show that they are capable of processing the underlying data stream multiple times faster. For Covertype *EvoAutoML* achieves on a processor with 2.5 GHz (Intel(R) Xeon(R) Platinum 8180M) a throughput of 36 Hz, *OB* 32 Hz and *LB* 38 instances per second. The single algorithms, however, are with 431 Hz (*HT*), 1447 Hz (*GNB*) and 903 Hz *k-NN* a multiple times faster than the ensemble techniques. Considering both accuracy and the time consumption, the *ARF* ensemble model returns with 89.7% accuracy and a throughput of 283 Hz, the best compromise between predictive performance and time consumption on Covertype. However, for a low-frequency data stream, *EvoAutoML* returns the highest accuracy score.

In Table 9.2, we depict the final percentage of instances correctly classified by the single as well as the *ensemble learning* techniques on various stream generators and the real-world data sets Covertype and Electricity. Furthermore, we measure the average rank over the data streams by comparing all evaluated models. Table 9.2 shows that *EvoAutoML* achieves with 2.08 the best rank averaged over all data streams.

Table 9.2: Accuracy comparison of *EvoAutoML* against baselines. Accuracy is measured as the final percentage of examples correctly classified. The best individual accuracies are indicated in boldface

Data Stream	EvoAutoML	HT	GNB	KNN	ARF	LB	OB
Agrawal(50)	99.02 ±0.01	98.09 ±0.01	62.31 ±0.09	55.73 ±0.02	94.98 ±0.95	**99.69** ±0.00	98.46 ±0.01
Agrawal(50000)	94.43 ±0.02	91.84 ±0.02	62.33 ±0.09	55.52 ±0.02	93.03 ±0.93	**97.52** ±0.01	92.89 ±0.02
HYP(50,0.0001)	**87.51** ±0.02	84.38 ±0.00	91.61 ±0.01	67.93 ±0.00	71.19 ±0.71	84.54 ±0.01	87.14 ±0.01
HYP(50,0.001)	83.69 ±0.01	81.79 ±0.01	80.83 ±0.02	68.01 ±0.00	71.67 ±0.72	**83.95** ±0.01	84.43 ±0.01
LED	**76.49** ±0.01	75.95 ±0.01	76.48 ±0.01	66.6 ±0.00	76.47 ±0.76	76.48 ±0.01	76.42 ±0.01
RBF(10,0.0001)	99.82 ±0.00	89.32 ±0.03	65.86 ±0.09	**100** ±0.00	99.85 ±0.01	99.64 ±0.00	98.07 ±0.00
RBF(10,0.001)	99.63 ±0.00	77.61 ±0.02	39.75 ±0.11	**99.99** ±0.00	99.22 ±0.99	99.01 ±0.00	93.68 ±0.01
RBF(50,0.0001)	97.51 ±0.01	83.05 ±0.03	35.26 ±0.13	**99.83** ±0.00	98.21 ±0.98	98.71 ±0.01	96.17 ±0.01
RBF(50,0.001)	96.99 ±0.01	48.15 ±0.04	25.32 ±0.07	**99.80** ±0.00	94.31 ±0.94	93.56 ±0.01	71.87 ±0.03
Sine	**99.87** ±0.00	99.63 ±0.01	93.62 ±0.00	98.75 ±0.00	99.74 ±0.01	99.68 ±0.00	99.77 ±0.01
SEA(50)	98.99 ±0.00	97.78 ±0.01	95.65 ±0.00	97.23 ±0.00	99.64 ±0.01	**99.67** ±0.01	98.34 ±0.01
Electricity	**88.09** ±0.01	79.61 ±0.02	72.87 ±0.03	79.53 ±0.01	87.79 ±0.88	87.32 ±0.01	81.74 ±0.02
Covertype	**91.09** ±0.07	66.67 ±0.10	63.64 ±0.11	73.74 ±0.12	89.7 ±0.09	90.41 ±0.08	83.66 ±0.12
Avg. Acc.	**93.32**	82.61	66.58	81.74	90.45	93.09	89.43
Avg. Rank	**2.08**	5.31	5.92	4.77	3.46	2.54	3.85

Furthermore, it achieves with 93.32% the best averaged percentage of instances correctly classified. Similar to Figure 9.1, *EvoAutoML* is followed by *LB* with an averaged rank ranging from of 2.54 and an accuracy of 93.09%. Again, Table 9.2 supports the results stated within the Covertype data set in Figure 9.1 in that the ensemble techniques perform with an average rank from 2.08 to 3.85 significantly better than the single algorithms (4.77 to 5.92). Comparing *EvoAutoML* against the single baseline approaches shows that *EvoAutoML* outperform the single algorithm by at least 10.71%. While the ensemble *LB* and *EvoAutoML* perform complainingly well. The *OB* and *ARF* ensembles achieve $\sim 3\%$ worse average accuracy than *EvoAutoML* or *LB*. To further depict the memory and time consumption, we present in Table 9.3 the

memory consumption (in Mb), as well as the avg. time required to process the data streams. To compute

Table 9.3: Comparison of memory consumption (in Mb) and avg. Time (in s) of *EvoAutoML* against different baselines. One RAM-Hour equals to 1 Gb of RAM deployed for 1 hour.

Data Stream	EvoAutoML	HT	GNB	KNN	ARF	LB	OB
Agrawal(50)	17.609	0.604	0.013	0.455	11.093	12.205	6.008
Agrawal(50000)	56.854	2.223	0.013	0.455	12.920	37.600	21.501
HYP(50,0.0001)	104.576	18.287	0.066	2.020	229.900	528.847	180.870
HYP(50,0.001)	127.877	18.516	0.066	2.020	356.600	395.203	187.146
LED()	35.954	2.104	0.048	0.379	10.133	39.723	18.570
RBF(10,0.0001)	24.527	13.359	0.133	2.020	25.803	22.897	134.988
RBF(10,0.001)	36.107	30.530	0.133	2.020	11.668	4.893	291.346
RBF(50,0.0001)	64.458	24.165	0.166	2.020	27.117	35.643	236.124
RBF(50,0.001)	29.288	9.173	0.166	2.020	25.453	8.023	98.340
Sine	9.760	0.421	0.004	0.169	14.622	11.128	4.211
SEA(50)	17.833	0.716	0.005	0.205	8.408	14.070	7.454
Electricity	12.697	0.205	0.012	0.417	6.850	1.729	1.938
Covertype	12.082	0.125	0.080	2.170	4.750	15.549	19.368
Avg. Time	33,638	4,635	1,489	2,119	56,786	58,347	35,243
RAM-Hours	7.19	0.32	0	0.01	50.38	44.55	24.35

the RAM-hours consumed by the underlying model, we multiply with a granularity of each $1,000^{th}$ step the memory with the time required. Whereby one RAM-hour equals to 1 Gb of RAM memory deployed for one hour. In accordance with Figure 9.1, Table 9.3 shows that the single algorithm approaches are a multiple faster in processing the data streams depicted in Section 9.3.1. While the single algorithms are in a magnitude of a few hours (< 5 h), the ensemble approaches require up to 16 h on average to process $1M$ instances within the data stream generators. Notably, the Hyperplane data stream with 50 features is computationally expensive. Comparing *EvoAutoML* with the other *ensemble learners*, it takes, on average, the least time. Considering the memory (*R I–3*) consumption a similar pattern is emerging as in the evaluation of the time consumption. The single algorithm learners consume a fraction of the memory compared to the ensemble learners. Combining the time required and the memory consumed within the employed RAM-hours, *EvoAutoML* requires for the *ensembles* with 7.19 RAM-hours the least resources and only a fraction of the RAM-hours. Concluding the results of this section, it emerges that the ensemble learners perform best concerning the final percentage of correctly classified instances. Including the time and memory requirements in the evaluation, it is shown that the single algorithms consume only a fraction of the resources in comparison to the *ensemble learners*. Within the *ensemble learners*, *EvoAutoML* achieved marginally better results than *LB*, *ARF* and *OB*, whereby it required the fewest resources when considering the time (*R I–2*) and memory (*R I–3*) requirements.

In summary, we show beside the requirements *R I–4* and *R I–1* in this section that *EvoAutoML* meets the requirements *R I–2*, *R I–3* and *R I–5* defined by [23] by consuming less time and memory as state-of-the-art ensemble learners and adapting to the underlying data patterns. We presented within *EvoAutoML* an approach that evolves the underlying *ML* pipelines configurations as the data stream evolves and thus answer RQ III.2 for *AutoML* by applying *EvoAutoML* on the *online ML* pipelines. Regarding RQ III.3 the predictive performance of *AutoML* increases slightly in comparison to ensemble learning approaches and significantly in comparison to single algorithms. Further, it consumes less time and memory than the compared *ensemble learning* techniques.

9.4 Online NAS

This section investigates the application of *NAS* in an *online learning* environment. As basis, we proposed within Chapter 8 an *online DL* framework and showed the suitability of *NN*s in *online learning* environments. In Section 9.2, we proposed in *EvoAutoML* an incremental *HPO* framework that configures underlying models and *ML* pipelines in an *online learning* manner. Within the previous section, we applied this framework on *ML* pipelines that are configured by a search space $(G, \mathcal{A}, \Lambda)$ and showed that the identically named *AutoML* (*EvoAutoML*) approach outperformed state-of-the-art *ensemble* models. Another outcome in this section is the greater consumption of computing resources of *ensemble learners*, whereby *EvoAutoML* required the least amount of time and the least amount of memory to process the data streams when considering only *ensemble learners*. In this section, however, we aim to perform *NAS* by combining the *online DL* and the *EvoAutoML* framework.

The structure of this section starts accordingly to Section 9.3 with the description of the experimental setup in Section 9.4.1 followed by the presentation of the results in Section 9.4.2.

9.4.1 Experimental Setup

The experimental setup follows the experiment design of the previous sections, whereby we choose a *test-than-train* evaluation protocol and similar data streams with $1M$ instances as in the earlier evaluations. Furthermore, we evaluate the accuracy and the time and memory efficiency. For the baseline models, we chose the identical single and *ensemble* learners as implemented for the evaluation of *EvoAutoML* in Section 9.3. The experimental setup is summarised in Table 9.4.

Table 9.4: Experimental Setup *EvoNAS*

Parameter	Setting
Stream Generators	Agrawal$(50, 000)$, Agrawal(50), Covertype, Electricity, HYP$(50, 0.0001)$, HYP$(50, 0.001)$, LED, RBF$(10, 0.0001)$, RBF$(10, 0.001)$, RBF$(50, 0.001)$, RBF$(10, 0.0001)$, *SEA*(50), Sine
Metric	Accuracy
	Memory consumption in Mb
	Time efficiency
Preprocessing	Standard Scaling
Models	
ARF	#models: 10
	max memory: 32 Mb
OB	#models: 10
	base model: *HT*
GNB	-
k-NN	#neighbors: 5
	window_size: 1000
HT	max memory: 32 Mb
EvoNAS	population: 10
	f_{SS}: 1000
	optimizer: *SGDHD*
	learning rate: 0.01
	architecture: *MLP*
	width: 1,2,3,10,15 or 20 neurons
	depth: 1,2,3,4,5 or 6 layers
	activation: *LeakyReLU*, *ReLU*, *Sigmoid*, *Softmax* or None

To combine the *EvoAutoML* framework and the *online DL* framework, the *EvoAutoML* estimator is initialised with a single algorithm type $A^{(i)}$; the *online DL* estimator. As depicted in Chapter 8, this estimator can be parametrised by various *NN* functions, optimisers or learning rates. We consider within our *NAS* approach

only structural changes within the *NN* and thus build the evaluation on a *MLP* search space. The *MLP* structure has the significant advantage that the structure can be relative to other architecture types such as *CNN* and *LSTM* relatively simple to be manipulated. The width of the *MLP* ranges from 1 to 20 (see Table 9.4) neurons and the *MLP*'s depth from 1 to 6 layers Each layer can be followed by a *LeakyReLU*, *ReLU*, *Sigmoid*, *Softmax* or no activation layer. The neural architecture's configuration space Λ can be set accordingly to the *Scikit-learns* design principles within a build function. The build function that is passed for initialisation to the *online DL* estimator is depicted in Listing 8.1. However, considering the search space of the underlying *MLP*, *EvoNAS* searches within $6 \times 6 \times 5 = 150$ different neural architectures.

Listing 9.1: MLP build function with variable depth, width and activation function

```
from torch import nn

def build_nn(n_features, depth, width, activation):
    modules = []
    modules.append(nn.Linear(n_features,width))
    for d in range(depth):
        modules.append(nn.Linear(width, width))
        if activation == 1:
            modules.append(nn.LeakyReLU())
        elif activation == 2:
            modules.append(nn.ReLU())
        elif activation == 3:
            modules.append(nn.Sigmoid())
        elif activation == 4:
            modules.append(nn.Softmax())
        else:
                pass

    modules.append(nn.Linear(width, 1))
    modules.append(nn.Sigmoid())
    net = nn.Sequential(modules)
    return net
```

In Listing 9.1, the *NN*'s configuration $(g, \overrightarrow{z}, \overrightarrow{\lambda})$ is created within a list ("modules"), whereby the architecture can vary in its depth (g), it its activation and thus in its node types $\overrightarrow{z}$. Each neural network of the *NAS* search space takes within the first layer the number of features (determined within the first instance of the data stream) and converts it to the *MLP*s width. Within each step of the loop over the depth of the *NN*, a new layer with the according activation is added to the structure of the *NN*. The *MLP*'s last layer summarises the width in one layer to the output of the *NN*, followed by a Sigmoid activation function. This output layer is accordingly to Algorithm 7 adapted to the different classes occurring within the data stream. The *MLP* structure, allows to show the capability to *EvoAutoML* to fully configure a *NN* by its structure g, its node types $\overrightarrow{z}$ and their configurations $\overrightarrow{\lambda}$. According to the results achieved within the evaluation of *online DL* in Section 8.3, the *NAS* optimiser searches in comparison to *offline learning NAS* search spaces (e.g. *NATS-Bench*) for relatively small *NN*s. Furthermore, this experimental setup answers research question *R III–2* technically in that it illustrates the simplicity towards the combination and configuration of the *EvoAutoML* and the *online DL* frameworks to a *online NAS* system.

9.4.2 Results

This section aims to answer research question RQ III.3 for the application of *NAS* in an *online learning* environment. Accordingly to the evaluation of *EvoAutoML*, we present in Figure 9.2 the accuracy curve and time consumption of the evaluated instances for *EvoNAS* against the baseline approaches on the Covertype data stream. It again shows the superiority in the predictive performance of the *ensemble* learners, but also their exhaustive time consumption.

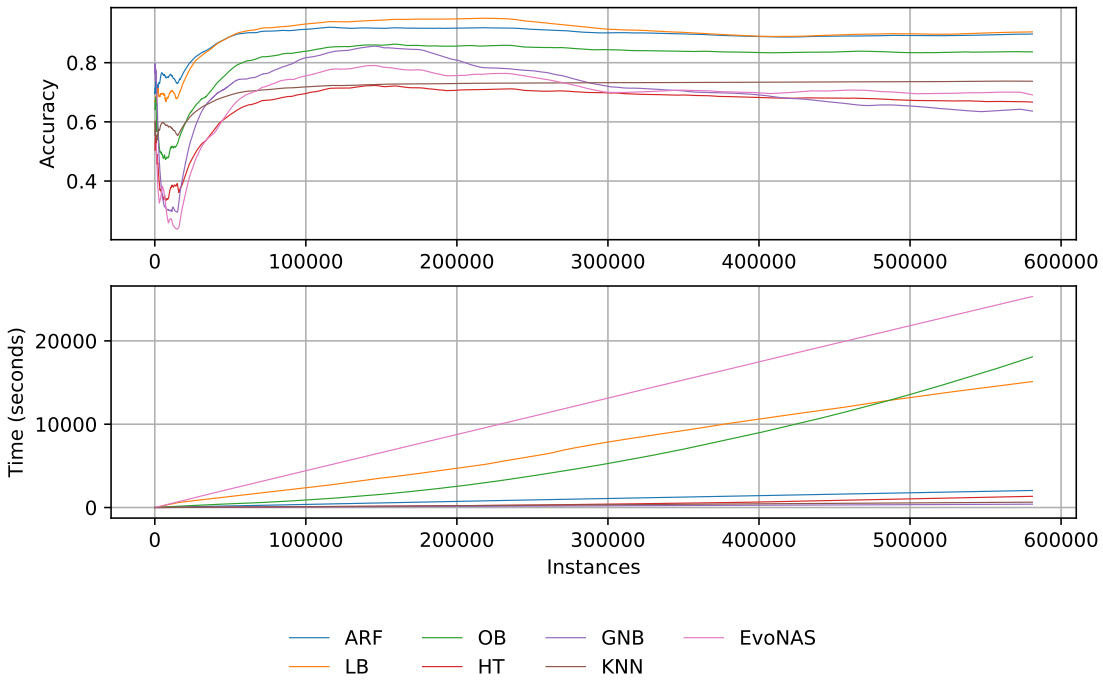


Figure 9.2: Accuracy curve and time (in seconds) for *EvoNAS* and baseline algorithms on Covertype.

In contrast to the evaluation of *EvoAutoML*, *EvoNAS* performs inferior on the evolving accuracy against the other *ensemble* learners and shows within the time plot the complexity of *NAS*. The time required to train ten *NN* in a parallel manner remains computational expensive. Comparing further the time plot of *EvoAutoML* in Figure 9.1 with the time consumption of *EvoNAS* it is noticeable, that the consumption of *EvoNAS* remains linear from the first instance, while the time consumption of *EvoAutoML* increases super-linear within the first instances and then increases linearly. While the linear increase of *EvoAutoML* is slower than the increase of the other *ensemble learners*, *EvoNAS* requires the most time to process the Covertype data stream. A similar results can be seen in Table 9.5 for the other data streams. Accordingly to the evaluation of *EvoAutoML*, we present in Table 9.5 the final percentage of examples correctly classified, the averaged final accuracy of all evaluated data streams and the average rank in comparison to the other approaches.

Table 9.5: Accuracy comparison of *EvoNAS* against baselines. Accuracy is measured as the final percentage of examples correctly classified. The best individual accuracies are indicated in boldface

Data Stream	EvoNAS	HT	GNB	KNN	ARF	LB	OB
Agrawal(50)	89.03 ±0.07	98.09 ±0.01	62.31 ±0.09	55.73 ±0.02	94.98 ±0.95	**99.69 ±0**	98.46 ±0.01
Agrawal(50,000)	88.07 ±0.06	91.84 ±0.02	62.33 ±0.09	55.52 ±0.02	93.03 ±0.93	**97.52 ±0.01**	92.89 ±0.02
HYP(50, 0.0001)	**94.58 ±0.05**	84.38 ±0	91.61 ±0.01	67.93 ±0	71.19 ±0.71	84.54 ±0.01	87.14 ±0.01
HYP(50,0.001)	**93.85 ±0.02**	81.79 ±0.01	80.83 ±0.02	68.01 ±0	71.67 ±0.72	83.95 ±0.01	84.43 ±0.01
LEDDrift()	73.95 ±0.11	75.95 ±0.01	76.48 ±0.01	66.6 ±0	76.47 ±0.76	**76.49 ±0.01**	76.42 ±0.01
RBF(10,0.0001)	98.83 ±0.07	89.32 ±0.03	65.86 ±0.09	**100 ±0**	99.85 ±1	99.64 ±0	98.07 ±0
RBF(10,0.001)	94.46 ±0.07	77.61 ±0.02	39.75 ±0.11	**99.99 ±0**	99.22 ±0.99	99.01 ±0	93.68 ±0.01
RBF(50,0.0001)	83.86 ±0.09	83.05 ±0.03	35.26 ±0.13	**99.83 ±0**	98.21 ±0.98	98.71 ±0.01	96.17 ±0.01
RBF(50,0.001)	43.82 ±0.05	48.15 ±0.04	25.32 ±0.07	**99.8 ±0**	94.31 ±0.94	93.56 ±0.01	71.87 ±0.03
SINE()	98.27 ±0.03	99.63 ±0.01	93.62 ±0	98.75 ±0	99.74 ±1	99.68 ±0	**99.77 ±0.01**
SEA(50)	99.57 ±0.03	97.78 ±0.01	95.65 ±0	97.23 ±0	99.64 ±1	**99.67 ±0.01**	98.34 ±0.01
Elec	79.78 ±0.07	79.61 ±0.02	72.87 ±0.03	79.53 ±0.01	**87.79 ±0.88**	87.32 ±0.01	81.74 ±0.02
Covtype	69.11 ±0.09	66.67 ±0.1	63.64 ±0.11	73.74 ±0.12	89.7 ±0.9	**90.41 ±0.08**	83.66 ±0.12
Avg. Acc.	85.17	82.61	66.58	81.74	90.45	**93.09**	89.43
Avg. Rank	4.23	4.92	5.92	4.62	2.85	**2.15**	3.31

Table 9.5 shows, that *EvoNAS* is with an averaged accuracy of 85.17% in average the weakest *ensemble learner*, but performs better than the best evaluated single algorithm, *HT* with 82.61%. Only on the Hyperplane data streams, *EvoNAS* shows the best performances against the single and the *ensemble learners*. The results show furthermore, that when not considering *EvoAutoML*, *LB* performs best on most data streams, followed by *ARF*. Comparing the results of *EvoNAS* (on $1M$ instances) with the predictive results achieved within the evaluation of the *online DL* framework (on $100,000$ instances), *EvoNAS* performs in average worse than the static or dynamic *MLP* classifiers. Within the experimental setup in Section 9.4.1, we defined a search space that configures regarding the complexity small *NN*s, with up to 6 layers (depth). *EvoNAS* mutates these architectures by changing accordingly to Real et al. [197] on parameter of a parents configuration. This child configuration is reinitialised and thus trained from scratch. The evaluation of the *online DL* framework showed in Figure 8.4, that the *MLP* configurations require more instances to learn the underlying data distribution. The slower adaptation to the data stream of *MLP* and the changing configuration within *EvoNAS* might lead to the lower predictive performance of *EvoNAS*. In Table 9.6, we depict the memory consumption accordingly to the evaluation of *EvoAutoML* in Section 9.3.

Table 9.6: Comparison of the avg. memory consumption (in Mb) and avg. time (in s) for *EvoNAS*. One RAM-Hour equals to 1 Gb of RAM deployed for 1 hour.

Data Stream	EvoNAS	HT	GNB	KNN	ARF	LB	OB
Agrawal(50)	0.541	0.604	0.013	0.455	11.093	12.205	6.008
Agrawal(50,000)	0.546	2.223	0.013	0.455	12.920	37.600	21.501
HYP(50, 0.0001)	0.496	18.287	0.066	2.020	229.900	528.847	180.870
HYP(50,0.001)	0.468	18.516	0.066	2.020	356.600	395.203	187.146
LED	0.480	2.104	0.048	0.379	10.133	39.723	18.570
RBF(10,0.0001)	0.644	13.359	0.133	2.020	25.803	22.897	134.988
RBF(10,0.001)	0.642	30.530	0.133	2.020	11.668	4.893	291.346
RBF(50,0.0001)	0.567	24.165	0.166	2.020	27.117	35.643	236.124
RBF(50,0.001)	0.509	9.173	0.166	2.020	25.453	8.023	98.340
Sine	0.489	0.421	0.004	0.169	14.622	11.128	4.211
SEA(50)	0.403	0.716	0.005	0.205	8.408	14.070	7.454
Electricity	0.451	0.205	0.012	0.417	6.850	1.729	1.938
Covertype	0.655	0.125	0.080	2.170	4.750	15.549	19.368
Avg. Time	62,564	4,635	1,489	2,119	56,786	58,347	35,243
RAM-Hours	3.64	0.32	0	0.01	50.38	44.55	24.35

Table 9.6 shows even clearer the computational complexity of *EvoNAS*. While the memory consumption in comparison with the other single algorithms remains comparable and in comparison with the *ensemble*

learners the consumption remains a multiple lower, the average time to process the data stream is the highest. In accordance with the evaluation of *online DL* and *EvoAutoML* these results are expected in that the evaluation showed already a comparable memory consumption and a lower throughput (time measure) for single *NN*s, *EvoNAS* multiplies this result, since it trains an ensemble of ten *NN*s. This results in the highest time consumption in comparison to the other approaches. Considering the consumed RAM-hours, *EvoNAS* requires a fraction of the RAM-hours compared to the *ensemble learners*. Due to the time required to process the underlying data stream of *EvoNAS* the required RAM-hours are a multiple times higher than the RAM-hours required by the single algorithms.

For the application of *EvoAutoML* in the context of *NAS*, we defined a search space that configures small *NN*s wrapped within the *online DL* estimator. Already small *MLP* architectures showed in Section 8.3 comparable results and substantiate the application of *NN*s in an *online learning* environment. The application of *NAS* with an underlying evolutionary *HPO*, however, showed inferior results concerning the predictive performance, but also the time and memory requirements for *online learning*. The results presented in this section show the applicability of *NAS* in an *online learning* environment, its complexity, and also new challenges. While in *offline learning* the search process is decoupled from the application of the *NN*, the application within *online learning* environments integrates the search for suitable architectures with the application. The idea similar to the application of *AutoML* on data streams was that the configuration evolves with the underlying data stream. The conflation of the search for neuronal architectures and the simultaneous use of found *NN*s, however, also shows that it negatively influences the latency and, thus, the throughput of the underlying algorithm. A further disadvantage is the slower convergence of larger *NN* (see. Figure 8.4), influences the performance of *EvoNAS* negatively in that it is not suitable A viable solution to this problem is the application of network morphisms; that reuse the weights of already trained architecture components and thus achieve a faster adaptation of the data distribution. Considering research question RQ III.2 towards the application of *NAS* we presented an approach to perform *NAS* with an underlying evolutionary approach on data streams. The presented approach showed, in comparison to the state-of-the-art approaches, inferior results considering the *online learning* requirements. However, the possibilities in varying the underlying optimisation process, using network morphisms or expanding the search space for other neural architectures, such as *LSTM*s, show a vast range of possibilities for the adaptation of data streams.

9.5 Summary

In this chapter, we introduced an incremental *HPO* framework that enables the automated configuration of *ML* pipelines (*AutoML*) and *NN*s (*NAS*). We first formalised accordingly to the *CASH* problem definition for *offline learning* an *online CASH* problem in Definition 20, presented the *EvoAutoML* framework that incrementally adapts configuration spaces based on an evolving population of algorithms and algorithm pipelines. The application of a *ML* pipeline search space showed that *EvoAutoML* is capable of outperforming state-of-the-art single learners, as well as *ensemble learners* in predictive performance. Regarding the memory and time requirements and thus the throughput of the algorithms, it emerged that *EvoAutoML* outperformed the *ensemble learner*. Still, the application of single algorithms requires a fraction of the resources for processing a data stream. The application of *NAS* on data streams showed that the proposed evolutionary approach does not lead to better performances in predictive accuracy but also in the time and memory requirements when executing on data streams. With both empirical evaluations for *AutoML* and *NAS*, we tested Hypothesis II and showed that the incremental adaption of hyperparameters enables better performances on data streams by following the *online learning* requirements for *AutoML* but not for *NAS*. Thus we can answer the research questions RQ III.2 and RQ III.3 in that we proposed with *EvoAutoML* an evolution-based approach to incrementally adapt a configuration space to data streams. Regarding the

performance *EvoAutoML* leads to better performances for the configuration of *ML* pipelines, but to inferior performances for *NAS*. The search for neural architectures under the *online learning* assumption leads to new challenges,For the incremental adaptation of *NAS*, To this end, we have introduced an *online DL* framework that merges the vast capabilities of *PyTorch* for *NN*s and the simple application of algorithms in an *online learning* environment. This framework fosters further research towards the application of *NAS*.

Part V

Synthesis

10 Conclusion and Outlook

This thesis investigated the adaptivity of *AutoML* and *NAS* systems towards certain utilities that go beyond the predictive performance and the adaptivity to data streams in *online learning* environments. Motivated by a traditional *KDD* process, we locate the adaptation towards a specific utility of *AutoML* and *NAS* techniques at the beginning and the adaptation to data streams at the end of the process, whereby *AutoML* aims to automate the steps in between by building a *ML* pipeline. Accordingly to the adaptivity of *AutoML*, we evaluate our approaches on the to *AutoML* related research field *NAS*, where the task is to search for suitable neural architectures. To this end, we provide in Part III a *LTR* system that learns an underlying metric to steer the search for suitable configurations of *ML* pipelines and neural architectures. And in Part IV, we present two frameworks, that aim to enable and simplify the application of *NN*s and *AutoML* systems in *online learning* environments by following the *Scikit-learn* design principles. This chapter summarises and synthesises our main findings regarding the adaptivity to a specific utility and the adaptivity to data streams. We conclude this thesis with an outline of promising future research topics.

10.1 Summary

Recent systems, incorporating the search for suitable neural architectures (*NAS*), have shown impressive results by concatenating and configuring *ML* pipelines and *NN*s on predefined objectives and data sets. However, it is often challenging to design objectives well-suited to the particular data and task of interest. Further, recent *AutoML* and *NAS* systems assume that all data are available at the beginning of the learning process and do not change over time. This assumption often contradicts the way data is produced. *IoT* and *IIoT* sensors, for example, continuously produce a vast amount of data, where the underlying data distribution might change due to changing environments over time, or real-time data analysis is required. The term "adaptive" in this thesis was thus manifold in that it aimed to (i) steer the search process of *AutoML* and *NAS* systems toward a specific utility (Part III) that goes beyond the predictive accuracy and (ii) to adapt the search process to data streams (Part IV) within an *online learning* environment.

Part II provides the foundations and the related work for this work. Starting from the *KDD* process, we build the concept of a *ML* pipeline and depict various *HPO* techniques that aim to automate based on the *CASH* problem the search for suitable pipeline configurations. To adapt this search process to a specific utility, we introduce *LTR* approaches that enable the estimation of a ranking for *ML* pipelines or neural architectures based on pointwise, pairwise and list-wise comparisons. For the adaptation of *AutoML* and *NAS*, we present within the foundation the *online learning* concept, the differences for the evaluation setup and the principal algorithms that provide incremental adaptation, but also *online learning* capabilities. Within the related work (Chapter 3), we depict the state-of-the-art approaches towards (i) *LTR*, incorporating *HGML* that includes human interfaces for *ML* applications, the related work for (ii) multi-objective *AutoML* and *NAS*, (ii) mphonline ensemble learning approaches, and finally approaches that include (iv) *online DL*.

Part III was driven by the research questions that asked for a system that adapts *AutoML* (RQ I) and *NAS* (RQ II) to a certain utility an end-user might pursue. We derived from the research questions Hypothesis I:

> *Existing approaches for* AutoML *and* NAS *aim efficiently maximising individual or sets of objectives $\mathcal{L}$. By variation of the target function $\mathcal{L}$, the output of* AutoML *and* NAS *systems can be adapted and tailored to the needs of the user and thus to a certain utility.*

Based on the *CASH* problem, we formalised and integrated the search for a suitable goal, also known as *metric learning*. To test Hypothesis I, we depicted, based on the formalisation, a *metric learning* system that incorporates pairwise rankings to learn the underlying preference and ordering of *ML* pipelines created by *AutoML* systems or neural architectures created by *NAS* systems. We divided the evaluation for *AutoML* and *NAS* into two parts. The first evaluation concerned the capability of learning an objective $\mathcal{L}$ by pairwise comparisons that express a utility an end-user might pursue. We assumed a predefined metric and showed within the evaluation that the underlying *RankNet* approach is capable of learning a utility metric within ~ 10 pairwise comparisons. For *AutoML* systems, we assumed a set of metrics that are based on the predictive performance of the underlying *ML* pipeline and thus calculated based on the prediction the model makes and the available ground truth label. For *NAS* systems, we assumed, due to the broad variability and thus complexity, utilities that incorporate the latency and the complexity of the underlying architecture and thus utilities that go beyond the predictive performance of the underlying neural architecture. We showed in both cases that a utility-based metric could be approximated within a few pairwise comparisons. The second evaluation for the *AutoML* and *NAS* systems concerned the impact of the learned metric on the performance regarding the underlying utility that is pursued. For *AutoML*, we based our approach on *TPOT*. We showed that the utility learned within ten pairwise comparisons can steer the optimisation process of *TPOT* in the direction of the underlying utility. Considering *NAS*, we based our evaluation on the *NATS-Bench* benchmark data set and compared our evolution-based optimisation approach against state-of-the-art multi-objective approaches. With the evaluation of the utility-based adaptation for *AutoML* and *NAS*, we showed that both are capable of steering the optimisation process toward a certain utility (Hypothesis I).

Part IV investigates the adaptation of *AutoML* and *NAS* to data streams. Based on the main research question of this part, whether *HPO* techniques can be applied in an |online learning environment and leads to better performances, we derived the following hypothesis

> *Considering an online learning environment, the incremental adaptation of hyperparameters enables better performances on data streams by following the online learning requirements [23].*

To test this hypothesis for *NAS* and *AutoML*, we provide two main frameworks in this part. In order to enable *NAS*, we provide a *online DL* framework, that combines the vast possibilities of the batch learning *DL* framework *PyTorch* and the simple *online learning API* of the *river* framework. This framework aims to unify and foster the research in *online learning* for *NN*s and to simplify the application and engineering of *DL* models on data streams. To follow the *Scikit-learn* design principles, we show the general suitability and competitiveness of *NN* with the evaluation of the smallest possible *NN*, the logistic regression and of smaller *MLP* architecture on established data streams and data stream generators. The second framework, *EvoAutoML*, introduced in this part, aims to enable the incremental adaptation of configuration spaces and thus the application of *AutoML* and *NAS* on data streams. Based on the assumption of possibly infinite and evolving data streams, it is based on an evolutionary strategy that develops a population of configuration candidates over the instances of the stream. As the *online DL* framework it follows the *Scikit-learn* design principles and the *river API*. We present on a small configuration space that *EvoAutoML* follows the requirements for *online learning* and is capable of generating *ML* pipelines in an ensemble that outperform

state-of-the-art approaches in predictive performance but also in memory and time consumption. In order to perform *NAS* on data streams, we combined the *EvoAutoML* and the *online DL* framework within a predefined *MLP* search space. The evaluation showed the complexity of applying *NAS* on data streams, as it achieved inferior performances and lower throughput of instances when evaluating the data streams. Considering Hypothesis II, we showed that the incremental adaptation of hyperparameters leads to better predictive and less computational requirements for *AutoML* when compared against other *ensemble learners*. This, however, does not apply to the application of *NAS*. The evaluation of the proposed systems and the frameworks are made publicly available on Github [1]

10.2 Discussion

In this section, we discuss the results of the utility and data stream based adaptation of *AutoML* and *NAS*. The results of the utility-based adaptation of *AutoML* and *NAS* showed that the pairwise ranking approach aims to learn an underlying utility within a few comparisons and steers the system in the direction of the utility. Within this *LTR* approach, we assumed predefined human feedback and based the evaluation on synthetic pairwise comparisons. A central motivation for this synthetic evaluation was to empirically prove that learned metrics can be utilised for improving the *AutoML* and *NAS* search process concerning predefined targets. Since a human evaluation is dependent on an adequate interface, it is essential to first develop and evaluate such an interface on its own. However, the evaluation of a utility-based system showed the influenceability of data-centric approaches and the complexity when incorporating multiple and diverse objectives. However, a limitation of our approach is the dependency on the learned objective on a set of predefined metrics, as it constrains the utility to weighted combinations of individual metrics. The set of metrics is easily extensible. It may require more comparisons to learn the underlying utility but has no computational influence on the search process than approaches that search a Pareto-frontier.

Within the results for the adaptation of *AutoML* and *NAS* systems to data streams, we propose two frameworks, the *online DL* and *EvoAutoML* framework. The *online DL* framework fosters research and simplifies the application of *NN*s in an *online learning* environment. While in an *offline learning* scenario, the architectural complexity increased over time (e.g. in image processing tasks) to solve more and more complex tasks, the application of *NN*s remained unsuitable in an *online learning* environment as the architectural complexity. The back-propagation optimisation process tended to infringe the requirements for an efficient and fast processing of data streams. Within this framework, we empirically show that architectural simple and lightweight neural architectures achieve comparable results on data streams to established single *online learning* algorithms. This finding fosters further research for *DL* architectures such as *RNN* (e.g. *LSTM*) or small *CNN* architectures and is further supported by the *online DL* framework that builds a connection between the established offline *DL* library *PyTorch* and the simple to use *API* of the *online learning* library *river*. A particular drawback of merging the *PyTorch* and the *river* library for straightforward development of neural architectures in *online learning* is the emerging overhead in complexity which inhibits the throughput of the data stream. The second framework, *EvoAutoML*, enables the application of incremental *HPO* techniques and thus the application of *AutoML* and *NAS* on data streams. It trains an ensemble of *ML* pipelines or neural architectures and evolves over a predefined configuration space by applying evolutionary strategies. *EvoAutoML* outperformed state-of-the-art ensemble learning algorithms, respectively, their predictive performance but also their memory and time consumption by searching for *online ML* pipelines consisting of preprocessing and prediction steps. This, however, did not apply to *NAS* as the training of multiple *NN*s requires a considerable amount of time to process an instance of the data

[1] https://github.com/kulbachcedric, last accessed January 30, 2023

stream. Further, within the evaluation of the *online DL* framework, it emerged that larger neural architectures require more instances to learn the underlying data pattern and thus adapt slower, which results in an inferior accuracy over the processed data streams.

Given the title *"Adaptive Automated Machine Learning"*, we presented in this thesis the adaptation of *AutoML* and *NAS* towards a utility that goes beyond the predictive performance and the adaptation of *AutoML* and *NAS* to data streams. The utility-based configuration of *ML* models and algorithm pipelines in *AutoML* or *NAS* during a data stream entails multiple difficulties in that the question towards the best suitable configuration gets accordingly to the utility adaptation of *AutoML* and *NAS* presented in Chapter 4 an additional time dimension. This additional dimension incorporates that the underlying data patterns may change within concept drifts of the data stream, but also the underlying utility might change due to changing environments.

10.3 Outlook

The utility-based adaptation of *AutoML* and *NAS* showed a fruitful direction to extend the search for suitable *ML* pipelines and neural architectures. A possible direction is to warm-start the utility-based system with preferences that other end-users pursued. Such a warm-start would also be directly applicable to the *Metric Learner* component of the proposed system, which could then converge with fewer samples. Further, the system boundaries of the proposed system enable to development of different user interfaces for various *ML* tasks. As *metric learning* incorporates approaches with and without (human) feedback, data-centric *metric learning* approaches that integrate the search for a suitable utility within the optimisation process of the underlying *ML* algorithm could be integrated within the search for suitable *AutoML* and *NAS* configurations. This, therefore, excludes the possibility of steering the optimisation process of the *AutoML* or *NAS* system more independently from the underlying data distribution. Within the *Evaluation Initiator*, we proposed, for the sake of simplicity, a random selection of The random choice of candidates could be extended in the form of active learning to gradually choose candidate comparisons that lead to faster convergence to the underlying utility.

Concerning the adaptation of *AutoML* and *NAS*, we build with the *EvoAutoML* and *online DL* frameworks a fruitful direction for further research. For the application of *AutoML* on data streams, the underlying optimisation process while the data stream is executing is crucial for the predictive performance, but also for the memory and time consumption and thus for the throughput of the algorithm. Furthermore, the search space could be further extended in that feature engineering algorithms are applied feature-wise. As some features might be categorical within the data stream, further encoding steps (e.g. one-hot-encoding) could be automatically applied to this data stream feature. Such an extension would massively increase the search space but also the capabilities of executing *AutoML* on data streams. While *NAS* showed inferior performances when considering the *online learning* requirements, an incremental *HPO* technique that uses network morphisms would accelerate the adaptation of *NN*s. An optimisation technique that does not rely on ensembles of *NN*s might further lead to better predictive performances but also to a better time and memory consumption. At this end, the proposed *online DL* framework enables the search for neural architectures that are suitable for *online learning*. Network architectures and parametrisations that work well on data streams can be discovered within the framework and are standardised for further use and applications. At this end, *DL* techniques showed not only impressive results for supervised learning but also for unsupervised learning tasks such as anomaly detection with auto-encoders. The detection of anomalies within the data streams is essential for the processing of, e.g. *IoT* and *IIoT* sensor data as it identifies malfunctioning data sources in

early stages and may protect downstream steps from significant damage. Here, the *online DL* framework will provide auto-encoded anomaly detection approaches that learn the underlying distribution of a data stream.

A Appendix

A.1 Results Online Deep Learning

This section provides additional results achieved within the proposed *online DL* framework. Table A.1 depicts the rolling accuracy with an window size of $1,000$ instances for a logistic regression classifier implemented within the *online DL* framework for different optimisers. And Table A.2 provides the rolling accuracies with a similar window size for the static *MLP* implementation regarding different optimisers.

Within Tables A.3 and A.4, we depict the results for the Logistic Regression and *MLP* models on different learning rates and data streams.

Table A.1: Rolling Accuracy comparison Torch Logistic Regression considering the optimisers *Adam*, *AdamW*, *SGD*, *SGDHD* and *RMSprop*. Accuracy is measured as the average rolling percentage of examples correctly classified.

Data Stream	**Torch Logistic Regression**				
	Adam	**AdamW**	**RMSprop**	**SGD**	**SGDHD**
Agrawal(50,000)	57.8 ±0.06	57.74 ±0.06	57.83 ±0.06	58.99 ±0.06	59.08 ±0.06
Agrawal(50)	62.2 ±0.09	62.14 ±0.09	62.18 ±0.09	63.39 ±0.09	63.5 ±0.09
Covertype	84.61 ±0.08	84.25 ±0.08	89.34 ±0.07	87.82 ±0.08	90.38 ±0.08
Electricity	83.48 ±0.05	82.96 ±0.06	87.99 ±0.03	85.46 ±0.04	88.54 ±0.03
HYP(50, 0.0001)	88.18 ±0.02	87.53 ±0.02	88.42 ±0.02	91.49 ±0.02	91.62 ±0.02
HYP(50,0.001)	88.2 ±0.02	87.52 ±0.02	88.41 ±0.02	91.3 ±0.02	91.37 ±0.02
LED	76.57 ±0.02	76.6 ±0.02	76.59 ±0.02	76.52 ±0.02	76.52 ±0.02
RBF(10,0.0001)	99.83 ±0	99.47 ±0	99.82 ±0	99.76 ±0	99.76 ±0
RBF(10,0.001)	98.73 ±0.01	97.6 ±0	98.45 ±0.01	98.95 ±0.01	97.52 ±0.01
SEA(50)	99.19 ±0.01	98.86 ±0.01	99.11 ±0.02	99.06 ±0.02	99.06 ±0.02
SEA(50000)	92.65 ±0.02	92.59 ±0.02	92.65 ±0.02	92.55 ±0.02	92.55 ±0.02
Sine	98.32 ±0.01	98.15 ±0.01	98.38 ±0	98.35 ±0.01	98.35 ±0.01
Avg. Accuracy	85.81	85.45	86.60	86.97	**87.35**
Avg. Rank	3.00	4.42	2.53	2.83	**2.42**

Table A.2: Rolling Accuracy comparison Torch Static *MLP* classifier considering the optimisers *Adam*, *AdamW*, *SGD*, *SGDHD* and *RMSprop*. Accuracy is measured as the average rolling percentage of examples correctly classified.

Data Stream	Torch Static *MLP* Classifier				
	Adam	**AdamW**	**RMSprop**	**SGD**	**SGDHD**
Agrawal(50,000)	75.72 ±0.06	76.84 ±0.05	74.95 ±0.06	74.91 ±0.05	75.22 ±0.06
Agrawal(50)	92.31 ±0.08	89.79 ±0.08	89.5 ±0.08	80.61 ±0.11	80.77 ±0.11
Covertype	76.25 ±0.13	80.66 ±0.11	84.52 ±0.12	87.08 ±0.14	83.23 ±0.14
Electricity	77 ±0.1	74.64 ±0.13	91.48 ±0.03	86.8 ±0.07	84.9 ±0.08
HYP(50, 0.0001)	92.53 ±0.03	89.62 ±0.02	92.42 ±0.03	92.32 ±0.06	92.43 ±0.06
HYP(50,0.001)	91.98 ±0.03	89.74 ±0.02	91.8 ±0.03	91.75 ±0.06	91.82 ±0.06
LED	71.94 ±0.05	72.38 ±0.05	70.6 ±0.05	72.58 ±0.09	72.84 ±0.1
RBF(10,0.0001)	99.65 ±0.01	99.53 ±0.02	99.51 ±0.02	95.75 ±0.08	96.07 ±0.08
RBF(10,0.001)	97.36 ±0.02	99.25 ±0.01	96.94 ±0.02	97.24 ±0.06	97.15 ±0.06
SEA(50)	98.7 ±0.01	97.8 ±0.01	98.67 ±0.01	98.42 ±0.04	98.49 ±0.04
SEA(50000)	93.41 ±0.02	92.82 ±0.02	93.31 ±0.02	92.92 ±0.04	92.92 ±0.04
Sine	98.77 ±0.01	97.79 ±0.01	98.8 ±0.01	98.63 ±0.03	98.63 ±0.03
Avg. Accuracy	88.80	88.41	90.21	89.08	**92.71**
Avg. Rank	3.08	3.58	2.83	3.50	**2.80**

Table A.3: Rolling Accuracy comparison of *PyTorch* and *river* Logistic Regression considering various learning rates. Accuracy is measured as the average rolling percentage of examples correctly classified. .

Data Stream	River Logistic Regression													
	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	**0.001**	**0.01**	**0.1**	**1**	**10**	**25**	**50**	**100**
Agrawal(50,000)	0.57 ±0.05	0.57 ±0.05	0.57 ±0.05	0.57 ±0.05	0.57 ±0.05	0.58 ±0.04	0.6 ±0.05	0.6 ±0.06	0.56 ±0.05	0.53 ±0.04	0.53 ±0.03	0.52 ±0.03	0.52 ±0.03	0.52 ±0.03
Agrawal(50)	0.59 ±0.06	0.59 ±0.06	0.59 ±0.06	0.59 ±0.06	0.59 ±0.06	0.6 ±0.05	0.63 ±0.06	0.64 ±0.09	0.6 ±0.09	0.56 ±0.07	0.55 ±0.06	0.55 ±0.06	0.54 ±0.06	0.54 ±0.06
Covertype	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09
Electricity	0.58 ±0.05	0.58 ±0.05	0.58 ±0.05	0.59 ±0.05	0.63 ±0.05	0.71 ±0.07	0.79 ±0.07	0.84 ±0.05	0.89 ±0.03	0.92 ±0.03	0.91 ±0.03	0.91 ±0.03	0.91 ±0.03	0.91 ±0.03
HYP(50, 0.0001)	0.5 ±0.02	0.5 ±0.01	0.51 ±0.01	0.57 ±0.05	0.81 ±0.11	0.92 ±0.06	0.94 ±0.03	0.93 ±0.02	0.87 ±0.02	0.84 ±0.02	0.84 ±0.02	0.84 ±0.02	0.84 ±0.02	0.84 ±0.02
HYP(50,0.001)	0.5 ±0.02	0.5 ±0.02	0.51 ±0.01	0.57 ±0.04	0.76 ±0.09	0.85 ±0.05	0.9 ±0.02	0.92 ±0.02	0.87 ±0.02	0.84 ±0.02	0.85 ±0.02	0.84 ±0.02	0.84 ±0.02	0.84 ±0.02
LED	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.11 ±0.02	0.14 ±0.02	0.14 ±0.02	0.14 ±0.02	0.14 ±0.01	0.13 ±0.01	0.12 ±0.01	0.12 ±0.01	0.11 ±0.01
RBF(10,0.0001)	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.29 ±0.03	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01
RBF(10,0.001)	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.16 ±0.01	0.28 ±0.03	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01	0.3 ±0.01
SEA(50)	0.67 ±0.09	0.67 ±0.09	0.67 ±0.09	0.67 ±0.09	0.72 ±0.08	0.87 ±0.1	0.97 ±0.05	0.99 ±0.02	0.99 ±0.02	0.98 ±0.02	0.94 ±0.03	0.91 ±0.05	0.88 ±0.06	0.86 ±0.08
SEA(50000)	0.66 ±0.04	0.66 ±0.04	0.66 ±0.04	0.66 ±0.04	0.68 ±0.03	0.84 ±0.09	0.92 ±0.04	0.93 ±0.02	0.93 ±0.02	0.92 ±0.02	0.89 ±0.02	0.87 ±0.02	0.85 ±0.02	0.83 ±0.03
Sine	0.54 ±0.02	0.54 ±0.02	0.54 ±0.02	0.55 ±0.02	0.74 ±0.12	0.94 ±0.08	0.98 ±0.02	0.98 ±0.01	0.98 ±0	0.98 ±0.01	0.97 ±0.01	0.97 ±0.01	0.96 ±0.01	0.96 ±0.01
Avg. Accuracy	0.44	0.44	0.44	0.45	0.51	0.59	0.64	**0.65**	0.64	0.63	0.62	0.61	0.61	0.60
Avg.Rank	10.50	10.25	9.67	9.08	8.33	6.25	3.33	**2.58**	3.67	5.17	6.58	7.67	7.92	9.25
Data Stream	**Torch Logistic Regression**													
	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	**0.001**	**0.01**	**0.1**	**1**	**10**	**25**	**50**	**100**
Agrawal(50,000)	0.53 ±0.08	0.53 ±0.08	0.53 ±0.08	0.53 ±0.08	0.53 ±0.08	0.59 ±0.06	0.61 ±0.06	0.59 ±0.06	0.55 ±0.04	0.53 ±0.03	0.53 ±0.04	0.53 ±0.03	0.53 ±0.03	0.53 ±0.03
Agrawal(50)	0.53 ±0.1	0.53 ±0.1	0.53 ±0.1	0.53 ±0.1	0.53 ±0.1	0.6 ±0.08	0.64 ±0.08	0.63 ±0.09	0.59 ±0.08	0.56 ±0.07	0.56 ±0.07	0.56 ±0.07	0.56 ±0.07	0.56 ±0.07
Covertype	0.03 ±0.09	0.03 ±0.09	0.03 ±0.09	0.58 ±0.24	0.7 ±0.14	0.77 ±0.12	0.83 ±0.11	0.88 ±0.08	0.91 ±0.07	0.91 ±0.07	0.91 ±0.07	0.91 ±0.07	0.91 ±0.07	0.91 ±0.07
Electricity	0.58 ±0.06	0.58 ±0.06	0.58 ±0.06	0.6 ±0.07	0.66 ±0.08	0.73 ±0.07	0.8 ±0.06	0.85 ±0.04	0.91 ±0.03	0.92 ±0.03	0.91 ±0.03	0.91 ±0.03	0.91 ±0.03	0.91 ±0.03
HYP(50, 0.0001)	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.6 ±0.09	0.88 ±0.1	0.93 ±0.05	0.94 ±0.02	0.91 ±0.02	0.85 ±0.02	0.84 ±0.02	0.84 ±0.02	0.84 ±0.02	0.84 ±0.02	0.84 ±0.02
HYP(50,0.001)	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.6 ±0.09	0.81 ±0.08	0.87 ±0.04	0.91 ±0.02	0.91 ±0.02	0.85 ±0.02	0.84 ±0.02	0.85 ±0.02	0.84 ±0.02	0.84 ±0.02	0.84 ±0.02
LED	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.11 ±0.03	0.37 ±0.14	0.7 ±0.13	0.76 ±0.06	0.77 ±0.02	0.76 ±0.02	0.67 ±0.02	0.66 ±0.02	0.66 ±0.02	0.66 ±0.02	0.66 ±0.02
RBF(10,0.0001)	0.15 ±0.01	0.15 ±0.01	0.44 ±0.1	0.76 ±0.14	0.9 ±0.09	0.98 ±0.03	0.99 ±0.01	1 ±0	1 ±0	1 ±0	1 ±0	1 ±0	1 ±0	1 ±0
RBF(10,0.001)	0.15 ±0.01	0.15 ±0.01	0.21 ±0.07	0.43 ±0.08	0.57 ±0.14	0.69 ±0.15	0.9 ±0.06	0.97 ±0.01	0.99 ±0.01	0.98 ±0.01	0.98 ±0.01	0.98 ±0.01	0.98 ±0.01	0.98 ±0.01
SEA(50)	0.33 ±0.09	0.33 ±0.09	0.33 ±0.09	0.33 ±0.09	0.61 ±0.22	0.89 ±0.12	0.97 ±0.06	0.99 ±0.02	0.99 ±0.01	0.99 ±0.01	0.99 ±0.01	0.99 ±0.01	0.99 ±0.01	0.99 ±0.01
SEA(50000)	0.34 ±0.04	0.34 ±0.04	0.34 ±0.04	0.34 ±0.04	0.61 ±0.2	0.88 ±0.11	0.92 ±0.05	0.93 ±0.02	0.93 ±0.02	0.91 ±0.02	0.91 ±0.02	0.91 ±0.02	0.91 ±0.02	0.91 ±0.02
Sine	0.54 ±0.02	0.54 ±0.02	0.54 ±0.02	0.54 ±0.02	0.79 ±0.14	0.95 ±0.08	0.98 ±0.03	0.98 ±0.01	0.98 ±0	0.98 ±0.01	0.98 ±0.01	0.98 ±0.01	0.98 ±0.01	0.98 ±0.01
Avg. Accuracy	0.36	0.36	0.39	0.50	0.66	0.80	0.85	**0.87**	0.86	0.85	0.84	0.84	0.84	0.84
Avg.Rank	11.25	11.25	11.00	10.42	9.08	6.50	4.42	3.58	**2.42**	4.42	6.33	7.33	6.50	6.08

Table A.4: Rolling Accuracy comparison of Static and Dynamic MLP classifer considering various learning rates. Accuracy is measured as the average rolling percentage of examples correctly classified.

Data Stream	Torch Static *MLP* Classifier													
	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	**0.001**	**0.01**	**0.1**	**1**	**10**	**25**	**50**	**100**
Agrawal(50,000)	0.53 ±0.08	0.53 ±0.08	0.53 ±0.08	0.53 ±0.08	0.53 ±0.08	0.57 ±0.05	0.66 ±0.06	0.75 ±0.05	0.63 ±0.1	0.52 ±0.03	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02	0.51 ±0.02
Agrawal(50)	0.53 ±0.1	0.53 ±0.1	0.53 ±0.1	0.53 ±0.1	0.53 ±0.1	0.58 ±0.07	0.72 ±0.11	0.81 ±0.11	0.81 ±0.16	0.54 ±0.04	0.52 ±0.03	0.52 ±0.03	0.52 ±0.03	0.52 ±0.03
Covertype	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.22 ±0.09	0.61 ±0.27	0.72 ±0.23	0.87 ±0.14	0.78 ±0.12	0.86 ±0.17	0.9 ±0.16	0.9 ±0.16	0.9 ±0.16	0.9 ±0.16
Electricity	0.58 ±0.06	0.58 ±0.06	0.58 ±0.06	0.58 ±0.06	0.58 ±0.06	0.58 ±0.06	0.73 ±0.11	0.87 ±0.07	0.74 ±0.12	0.81 ±0.02	0.85 ±0.02	0.85 ±0.02	0.85 ±0.02	0.85 ±0.02
HYP(50, 0.0001)	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.54 ±0.03	0.85 ±0.15	0.92 ±0.06	0.85 ±0.06	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02
HYP(50,0.001)	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.53 ±0.03	0.83 ±0.16	0.92 ±0.06	0.84 ±0.05	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02
LED	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.2 ±0.07	0.56 ±0.17	0.73 ±0.09	0.16 ±0.09	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01
RBF(10,0.0001)	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.24 ±0.1	0.55 ±0.15	0.9 ±0.15	0.96 ±0.08	0.57 ±0.18	0.32 ±0.02	0.29 ±0.01	0.29 ±0.01	0.29 ±0.01	0.29 ±0.01
RBF(10,0.001)	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.22 ±0.1	0.34 ±0.07	0.88 ±0.16	0.97 ±0.06	0.48 ±0.12	0.32 ±0.02	0.29 ±0.01	0.29 ±0.01	0.29 ±0.01	0.29 ±0.01
SEA(50)	0.33 ±0.09	0.33 ±0.09	0.33 ±0.09	0.33 ±0.09	0.36 ±0.12	0.71 ±0.14	0.96 ±0.11	0.98 ±0.04	0.98 ±0.02	0.6 ±0.08	0.57 ±0.06	0.57 ±0.06	0.57 ±0.06	0.57 ±0.06
SEA(50000)	0.34 ±0.04	0.34 ±0.04	0.34 ±0.04	0.34 ±0.04	0.36 ±0.08	0.69 ±0.13	0.9 ±0.09	0.93 ±0.04	0.93 ±0.02	0.58 ±0.04	0.56 ±0.03	0.56 ±0.03	0.56 ±0.03	0.56 ±0.03
Sine	0.54 ±0.02	0.54 ±0.02	0.54 ±0.02	0.54 ±0.02	0.54 ±0.02	0.85 ±0.15	0.97 ±0.07	0.99 ±0.03	0.98 ±0.01	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02
Avg. Accuracy	0.37	0.37	0.37	0.37	0.39	0.56	0.81	**0.89**	0.73	0.51	0.51	0.51	0.51	0.51
Avg.Rank	8.67	8.67	8.67	8.67	8.33	4.75	3.42	**1.42**	3.25	6.42	7.00	7.75	7.33	7.33
Data Stream	**Torch Dynamic *MLP*Classifier**													
	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	**0.001**	**0.01**	**0.1**	**1**	**10**	**25**	**50**	**100**
Agrawal(50,000)	0.53 ±0.08	0.53 ±0.08	0.53 ±0.08	0.53 ±0.08	0.53 ±0.08	0.56 ±0.06	0.57 ±0.06	0.75 ±0.06	0.74 ±0.04	0.53 ±0.03	0.53 ±0.04	0.53 ±0.03	0.53 ±0.03	0.53 ±0.03
Agrawal(50)	0.53 ±0.1	0.53 ±0.1	0.53 ±0.1	0.53 ±0.1	0.53 ±0.1	0.58 ±0.08	0.59 ±0.08	0.89 ±0.09	0.79 ±0.08	0.53 ±0.07	0.53 ±0.07	0.53 ±0.07	0.53 ±0.07	0.53 ±0.07
Covertype	0.03 ±0.09	0.03 ±0.09	0.03 ±0.09	0.03 ±0.24	0.5 ±0.14	0.66 ±0.12	0.82 ±0.11	0.91 ±0.08	0.04 ±0.07	0.03 ±0.07	0.03 ±0.07	0.03 ±0.07	0.03 ±0.07	0.03 ±0.07
Electricity	0.58 ±0.06	0.58 ±0.06	0.58 ±0.06	0.58 ±0.07	0.58 ±0.08	0.58 ±0.07	0.74 ±0.06	0.87 ±0.04	0.6 ±0.03	0.44 ±0.03	0.42 ±0.03	0.42 ±0.03	0.42 ±0.03	0.42 ±0.03
HYP(50, 0.0001)	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.09	0.5 ±0.1	0.58 ±0.05	0.87 ±0.02	0.92 ±0.02	0.54 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02
HYP(50,0.001)	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.09	0.5 ±0.08	0.58 ±0.04	0.86 ±0.02	0.91 ±0.02	0.55 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02	0.5 ±0.02
LED	0.1 ±0.01	0.1 ±0.01	0.1 ±0.01	0.1 ±0.03	0.1 ±0.14	0.12 ±0.13	0.61 ±0.06	0.74 ±0.02	0.35 ±0.02	0.1 ±0.02	0.1 ±0.02	0.1 ±0.02	0.1 ±0.02	0.1 ±0.02
RBF(10,0.0001)	0.15 ±0.01	0.15 ±0.01	0.15 ±0.1	0.15 ±0.14	0.24 ±0.09	0.68 ±0.03	0.97 ±0.01	0.99 ±0	0.28 ±0	0.15 ±0	0.15 ±0	0.15 ±0	0.15 ±0	0.15 ±0
RBF(10,0.001)	0.15 ±0.01	0.15 ±0.01	0.15 ±0.07	0.15 ±0.08	0.23 ±0.14	0.37 ±0.15	0.95 ±0.06	0.99 ±0.01	0.18 ±0.01	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01	0.15 ±0.01
SEA(50)	0.33 ±0.09	0.33 ±0.09	0.33 ±0.09	0.33 ±0.09	0.62 ±0.22	0.66 ±0.12	0.89 ±0.06	0.98 ±0.02	0.94 ±0.01	0.34 ±0.01	0.33 ±0.01	0.33 ±0.01	0.33 ±0.01	0.33 ±0.01
SEA(50000)	0.34 ±0.04	0.34 ±0.04	0.34 ±0.04	0.34 ±0.04	0.61 ±0.2	0.66 ±0.11	0.84 ±0.05	0.92 ±0.02	0.92 ±0.02	0.34 ±0.02	0.34 ±0.02	0.34 ±0.02	0.34 ±0.02	0.34 ±0.02
Sine	0.54 ±0.02	0.54 ±0.02	0.54 ±0.02	0.54 ±0.02	0.54 ±0.14	0.54 ±0.08	0.71 ±0.03	0.96 ±0.01	0.98 ±0	0.49 ±0.01	0.46 ±0.01	0.46 ±0.01	0.46 ±0.01	0.46 ±0.01
Avg. Accuracy	0.36	0.36	0.36	0.36	0.46	0.55	0.78	**0.90**	0.58	0.34	0.34	0.34	0.34	0.34
Avg.Rank	6.50	6.50	6.50	6.42	5.00	3.58	2.42	**1.17**	3.00	9.92	10.42	10.00	9.75	8.58

List of Figures

List of Tables

List of Abbreviations

A

AdaGrad *Adaptive Gradient Algorithm* 30, 58
Adam *Adaptive Moment Estimation* 30, 58, 75, 76, 121, 125, 126, 129, 157, 158
ADWIN *Adaptive Window* 56, 58, 59, 74–76, 139
AutoML *Automated Machine Learning* 3–9, 11, 15–18, 21, 23, 31, 32, 35–45, 47–49, 52, 54, 57, 60, 61, 64, 66, 67, 70–72, 74, 76, 79–83, 85, 86, 88–97, 99, 100, 102–105, 107, 111, 113–115, 119, 135–139, 142, 143, 147, 151–154, 161, 163
API *Application Programming Interface* 10, 112, 113, 115, 120, 121, 123, 135, 152, 153
ARF *Adaptive Random Forest* 74, 136, 139–143, 146
ART *Adaptive Resonance Theory* 75
ATM *Auto-Tuned Models* 40, 42

B

BO *Baysian Optimization* 34, 37, 38, 41, 45, 47, 48, 71, 72
BOHB *Robust and Efficient Hyperparameter Optimization at Scale* 47, 48

C

CASH *Combined Algorithm and Hyperparameter optimization* 32, 39–43, 48, 64, 70, 79, 81, 135, 136, 147, 151, 152
CIFAR *Canadian Istitute For Advanced Research* 42, 45, 71, 102, 103, 105–107, 115, 161
CMA *Cummulative Moving Average* 63
CMA-ES *Covariance Matrix Adaption Evolution Strategy* 36
CNN *Convolutional Neural Network* 28, 29, 44, 45, 47, 48, 58, 72, 75, 76, 86, 87, 121, 144, 153
CRISP-DM *CRoss Industry Standard Process for Data Mining* 3–7, 16–18, 32, 161, 163

D

DAG *Directed Arbitrary Graph* 20–22, 29, 39, 43, 136
DDM *Drift Detection Method* 56
DL *Deep Learning* 3, 4, 10, 11, 28, 40, 42, 47, 48, 67, 74–76, 113–115, 119, 120, 123, 124, 130, 132, 135, 140, 143, 144, 146–148, 151–155, 157
DM *Data Mining* 3–5, 15–18, 52, 53, 56, 66, 111
DNN *Deep Neural Network* 28, 30, 47

E

EA *Evolutionary Algorithm* 35–38, 42, 45, 46, 48, 68, 71, 72, 161
EDDM *Early Drift Detection Method* 56, 74
EMD *Empirical Mode Decomposition* 75
EP *Evolutionary Programming* 35
ES *Evolutionary Strategy* 35, 36, 52, 69, 83, 99, 101
EvoAutoML *Evolution-based Online Automated Machine Learning* 10, 11, 131, 137–148, 152–154, 161, 163
EvoNAS *Evolution-based Neural Architecture Search* 10, 143–147, 161, 163

F

FLOPS *FLoating point Operations Per Second* 72, 103, 104, 107
FN *False Negative* 61
FNN *Feed Forward Neural Network* 26, 28, 29, 44, 72, 75, 76, 121
FP *False Positive* 61, 62, 71

G

GA *Genetic Algorithm* 34, 35, 136–138
GNB *Gaussian Naive Bayes* 127, 130, 132, 139, 141, 143
GP *Genetic Programming* 35, 40, 41, 48, 71
GPU *Graphical Processing Unit* 30, 43, 45, 70, 90, 122, 124, 139

H

HAT *Hoeffding Adaptive Tree* 57, 58, 131, 136
HBP *Hedge Backpropagation* 75
HDDM *Hoeffding's Drift Detection Method* 56
HGML *Human Guided Machine Learning* 5, 67–70, 79, 80, 107, 151, 163

HPO *Hypterparamter optimization* 8–10, 15, 18, 23, 32–34, 36–42, 46, 48, 52, 89, 99, 114, 115, 126, 135, 143, 147, 151–154
HT *Hoeffding Tree* 57, 58, 73, 127, 130–132, 139, 141, 143, 146

I

IID *Independent and Identically Distributed* 64
IIoT *Industrial Internet of Things* 3–6, 56, 151, 154
IoT *Internet of Things* 3–6, 56, 151, 154

K

KDD *Knowledge Discovery in Databases* 3, 4, 11, 15–20, 32, 151, 161, 163
k-NN *k-Nearest Neighbors* 25, 59, 68, 69, 73, 74, 79, 139, 141, 143

L

LB *Leveraging Bagging* 74, 135, 141, 142, 146
LSTM *Long-Short Term Memory* 75, 76, 121, 144, 147, 153
LTR *Learning To Rank* 49, 50, 86–88, 90, 92, 102, 105–107, 151, 153

M

MAE *Mean Absolute Error* 62, 63
MAPE *Mean Average Percentage Error* 63
ML *Machine Learning* 3–5, 7–11, 15, 17–23, 28, 31, 32, 34, 38, 39, 41, 42, 47–50, 52–54, 56, 59, 60, 63, 66–72, 76, 79, 80, 82, 83, 88–90, 92, 94, 96, 97, 99, 100, 111–114, 119, 135–140, 142, 143, 147, 148, 151–154, 161
MLP *Multi-Layer Perceptron* 26, 28, 29, 44–48, 58, 71, 73, 75, 76, 87, 121, 124–132, 143, 144, 146, 147, 152, 153, 157, 158, 160, 163
MOA *Massive Online Analysis* 111, 112, 119
MSE *Mean Squared Error* 62, 63, 92, 94, 96
MSTD *Moving Standard Deviation* 57

N

NAS *Neural Architecture Search* 4–11, 15, 16, 18, 26, 29, 32, 34, 36–38, 40–49, 52, 60, 61, 64, 66, 70–72, 74, 76, 79–83, 86, 88, 99–105, 107, 111, 115, 119, 121, 123, 124, 127, 135, 137, 138, 143–145, 147, 148, 151–154, 161, 163
NB *Naïve Bayes* 25, 26, 59, 73
NLP *Natural Language Processing* 45, 87
NN *Neural Network* 3, 6, 8–11, 18, 22, 23, 26, 28–30, 32, 43–47, 49, 51, 52, 57–59, 67, 74–76, 86, 99, 100, 102–105, 114, 115, 119–124, 126–128, 130–133, 135, 137, 138, 143–148, 151–154, 161, 163
NSGA-II *Nondominated Sorting Genetic Approach* 36

O

OB *Online Bagging* 73, 74, 135, 136, 139–143

P

PCA *Principal Component Analysis* 22, 68, 69
PIL *Parameter Incremental Learning* 75
PSO *Particle Swarm Optimization* 35

R

RBF *Radial Bias Function* 75, 117, 125
ReLU *Rectified Linear Unit* 27, 87, 143, 144
RL *Reinforcement Learning* 9, 26, 42, 45–47, 71, 102
RMSProp *Root Mean Squared Propagation* 30, 125, 126, 130
RNN *Recurrent Neural Networks* 28, 47, 58, 72, 121, 153

S

SEA *Streaming Ensemble Algorithm* 116, 124, 125, 127–130, 139, 143, 161
SEMMA *Sample, Explore, Modify, Model, Assess* 16–18, 32, 161, 163
SGD *Stochastic Gradient Descent* 30, 58, 75, 76, 121, 125–127, 129, 130, 143, 157, 158
SMAC *Sequential Model-based Algorithm Configuration* 38, 41, 42, 47, 48, 72
SNN *Spiking Neural Network* 75
SRP Streaming Random Patches 135
SVM *Support Vector Machine* 21, 24, 58, 68, 69

T

TanH *Hyperbolic Tangent* 27, 28
TN *True Negative* 61
TP *True Positive* 61, 71
TPOT *Tree-based Pipeline Optimization Tool* 9, 36, 40–42, 46–48, 71, 89, 90, 92, 104, 115, 152
TPU *Tensor Processing Unit* 70, 122

Bibliography

[1] Martín Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton; Timothy Roscoe. USENIX Association, 2016, pp. 265–283.

[2] Rakesh Agrawal; Tomasz Imielinski; Arun N. Swami. “Database Mining: A Performance Perspective”. In: *IEEE Trans. Knowl. Data Eng.* 5.6 (1993), pp. 914–925. DOI: 10.1109/69.250074.

[3] Alireza Akhbardeh; Sakari Junnila; Teemu Koivistoinen; Alpo Värri. “An Intelligent Ballistocardiographic Chair Using a Novel SF-ART Neural Network and Biorthogonal Wavelets”. In: *Journal of Medical Systems* 31.1 (Dec. 2006), pp. 69–77. ISSN: 0148-5598, 1573-689X. DOI: 10.1007/s10916-006-9044-x.

[4] Saad Albawi; Tareq Abed Mohammed; Saad Al-Zawi. “Understanding of a Convolutional Neural Network”. In: *2017 International Conference on Engineering and Technology (ICET)*. Ieee. 2017, pp. 1–6.

[5] Andrew Anderson; Jing Su; Rozenn Dahyot; David Gregg. “Performance-Oriented Neural Architecture Search”. In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 177–184.

[6] Ana Azevedo; Manuel Filipe Santos. “KDD, SEMMA and CRISP-DM: a parallel overview”. In: *IADIS European Conference on Data Mining 2008, Amsterdam, The Netherlands, July 24-26, 2008. Proceedings*. Ed. by Ajith Abraham. IADIS, 2008, pp. 182–185.

[7] Nurzety A. Azuan; Suzanne M. Embury; Norman W. Paton. “Observing the Data Scientist: Using Manual Corrections As Implicit Feedback”. In: *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2017, Chicago, IL, USA, May 14, 2017*. Ed. by Carsten Binnig; Joseph M. Hellerstein; Aditya G. Parameswaran. ACM, 2017, 13:1–13:6. DOI: 10.1145/3077257.3077272.

[8] Manuel Baena-Garcıa et al. “Early drift detection method”. In: *Fourth international workshop on knowledge discovery from data streams*. Vol. 6. 2006, pp. 77–86.

[9] Maroua Bahri; Bruno Veloso; Albert Bifet; Joao Gama. “AutoML for Stream K-Nearest Neighbors Classification”. In: *2020 IEEE International Conference on Big Data (Big Data)*. Atlanta, GA, USA: IEEE, Dec. 2020, pp. 597–602. ISBN: 978-1-72816-251-5. DOI: 10.1109/BigData50022.2020.9378396.

[10] Maroua Bahri et al. “Data stream analysis: Foundations, major tasks and tools”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 11.3 (2021), e1405.

[11] Bowen Baker; Otkrist Gupta; Nikhil Naik; Ramesh Raskar. “Designing Neural Network Architectures Using Reinforcement Learning”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[12] G Ball et al. “An Integrated Approach Utilizing Artificial Neural Networks and SELDI Mass Spectrometry for the Classification of Human Tumours and Rapid Identification of Potential Biomarkers”. In: *Bioinformatics (Oxford, England)* 18.3 (2002), pp. 395–404.

[13] Friedrich L Bauer. “Computational Graphs and Rounding Error”. In: *SIAM Journal on Numerical Analysis* 11.1 (1974), pp. 87–96.

[14] Atilim Gunes Baydin et al. “Online Learning Rate Adaptation with Hypergradient Descent”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[15] Aurélien Bellet; Amaury Habrard; Marc Sebban. “A Survey on Metric Learning for Feature Vectors and Structured Data”. In: *CoRR* abs/1306.6709 (2013). arXiv: 1306.6709.

[16] Richard Bellman. *Adaptive Control Processes - A Guided Tour (Reprint from 1961)*. Vol. 2045. Princeton Legacy Library. Princeton University Press, 2015. ISBN: 978-1-4008-7466-8. DOI: 10.1515/9781400874668.

[17] Moshe Ben-Bassat. “Use of Distance Measures, Information Measures and Error Bounds in Feature Evaluation”. In: *Handbook of Statistics*. Vol. 2. Elsevier, 1982, pp. 773–791. ISBN: 978-0-444-86217-4. DOI: 10.1016/S0169-7161(82)02038-0.

[18] Yoshua Bengio; Patrice Simard; Paolo Frasconi. “Learning Long-Term Dependencies with Gradient Descent Is Difficult”. In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[19] James Bergstra; Yoshua Bengio. “Random Search for Hyper-Parameter Optimization”. In: *J. Mach. Learn. Res.* 13 (2012), pp. 281–305.

[20] James Bergstra; Daniel Yamins; David Cox. “Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures”. In: *International Conference on Machine Learning*. PMLR. 2013, pp. 115–123.

[21] Albert Bifet; Ricard Gavaldà. “Adaptive Learning from Evolving Data Streams”. In: *Advances in Intelligent Data Analysis VIII, 8th International Symposium on Intelligent Data Analysis, IDA 2009, Lyon, France, August 31 - September 2, 2009. Proceedings*. Ed. by Niall M. Adams; Céline Robardet; Arno Siebes; Jean-François Boulicaut. Vol. 5772. Lecture Notes in Computer Science. Springer, 2009, pp. 249–260. DOI: 10.1007/978-3-642-03915-7_22.

[22] Albert Bifet; Ricard Gavaldà. “Learning from Time-Changing Data with Adaptive Windowing”. In: *Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA*. SIAM, 2007, pp. 443–448. DOI: 10.1137/1.9781611972771.42.

[23] Albert Bifet; Ricard Gavaldà; Geoff Holmes; Bernhard Pfahringer. *Machine Learning for Data Streams with Practical Examples in MOA*. MIT Press, 2018.

[24] Albert Bifet; Geoffrey Holmes; Bernhard Pfahringer. “Leveraging Bagging for Evolving Data Streams”. In: *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML PKDD 2010, Barcelona, Spain, September 20-24, 2010, Proceedings, Part I*. Ed. by José L. Balcázar; Francesco Bonchi; Aristides Gionis; Michèle Sebag. Vol. 6321. Lecture Notes in Computer Science. Springer, 2010, pp. 135–150. DOI: 10.1007/978-3-642-15880-3_15.

[25] Albert Bifet et al. “MOA: Massive Online Analysis, a Framework for Stream Classification and Clustering”. In: *Proceedings of the First Workshop on Applications of Pattern Analysis, WAPA 2010, Cumberland Lodge, Windsor, UK, September 1-3, 2010*. Ed. by Tom Diethe; Nello Cristianini; John Shawe-Taylor. Vol. 11. JMLR Proceedings. JMLR.org, 2010, pp. 44–50.

[26] Albert Bifet et al. “New Ensemble Methods for Evolving Data Streams”. In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD ’09*. Paris, France: ACM Press, 2009, p. 139. ISBN: 978-1-60558-495-9. DOI: 10.1145/1557019.1557041.

[27] Albert Bifet et al. “Pitfalls in Benchmarking Data Stream Classification and How to Avoid Them”. In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part I*. Ed. by Hendrik Blockeel; Kristian Kersting; Siegfried Nijssen; Filip Zelezný. Vol. 8188. Lecture Notes in Computer Science. Springer, 2013, pp. 465–479. DOI: 10.1007/978-3-642-40988-2_30.

[28] Jock A Blackard; Denis J Dean. “Comparative Accuracies of Artificial Neural Networks and Discriminant Analysis in Predicting Forest Cover Types from Cartographic Variables”. In: *Computers and electronics in agriculture* 24.3 (1999), pp. 131–151.

[29] Isvani Inocencio Frías Blanco et al. “Online and Non-Parametric Drift Detection Methods Based on Hoeffding’s Bounds”. In: *IEEE Trans. Knowl. Data Eng.* 27.3 (2015), pp. 810–823. DOI: 10.1109/TKDE.2014.2345382.

[30] Philip Bohannon; Wenfei Fan; Michael Flaster; Rajeev Rastogi. “A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification”. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data - SIGMOD ’05*. Baltimore, Maryland: ACM Press, 2005, p. 143. ISBN: 978-1-59593-060-6. DOI: 10.1145/1066157.1066175.

[31] Mohammad Reza Bonyadi; Zbigniew Michalewicz. “Particle Swarm Optimization for Single Objective Continuous Space Problems: A Review”. In: *Evol. Comput.* 25.1 (2017), pp. 1–54. DOI: 10.1162/EVCO_r_00180.

[32] Bernhard E Boser; Isabelle M Guyon; Vladimir N Vapnik. “A Training Algorithm for Optimal Margin Classifiers”. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. 1992, pp. 144–152.

[33] Nadia Boukhelifa; Anastasia Bezerianos; Evelyne Lutton. “Evaluation of interactive machine learning systems”. In: *Human and machine learning - visible, explainable, trustworthy and transparent*. Ed. by Jianlong Zhou; Fang Chen. Human-computer interaction series. tex.bibsource: dblp computer science bibliography, https://dblp.org tex.biburl: https://dblp.org/rec/series/hci/BoukhelifaBL18.bib tex.timestamp: Mon, 26 Oct 2020 08:19:22 +0100. Springer, 2018, pp. 341–360. DOI: 10.1007/978-3-319-90403-0_17.

[34] Nadia Boukhelifa; Anastasia Bezerianos; Waldo Cancino Ticona; Evelyne Lutton. “Evolutionary Visual Exploration: Evaluation of an IEC Framework for Guided Visual Search”. In: *Evol. Comput.* 25.1 (2017), pp. 55–86. DOI: 10.1162/EVCO_a_00161.

[35] George EP Box. “Robustness in the Strategy of Scientific Model Building”. In: *Robustness in Statistics*. Elsevier, 1979, pp. 201–236.

[36] Leo Breiman. “Bagging Predictors”. In: *Machine Learning* 24.2 (Aug. 1996), pp. 123–140. ISSN: 0885-6125, 1573-0565. DOI: 10.1007/BF00058655.

[37] Eric Brochu; Vlad M Cora; Nando De Freitas. “A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning”. In: *arXiv preprint arXiv:1012.2599* (2010). arXiv: 1012.2599.

[38] Andrew Brock; Theodore Lim; James M Ritchie; Nick Weston. “Smash: One-Shot Model Architecture Search through Hypernetworks”. In: *arXiv preprint arXiv:1708.05344* (2017). arXiv: 1708.05344.

[39] Greg Brockman et al. “Openai Gym”. In: *arXiv preprint arXiv:1606.01540* (2016). arXiv: 1606.01540.

[40] Nicholas J. Bryan; Gautham J. Mysore; Ge Wang. "ISSE: an interactive source separation editor". In: *CHI Conference on Human Factors in Computing Systems, CHI'14, Toronto, ON, Canada - April 26 - May 01, 2014*. Ed. by Matt Jones; Philippe A. Palanque; Albrecht Schmidt; Tovi Grossman. ACM, 2014, pp. 257–266. DOI: 10.1145/2556288.2557253.

[41] Dariusz Brzezinski; Jerzy Stefanowski. "Combining block-based and online methods in learning ensembles from concept drifting data streams". In: *Inf. Sci.* 265 (2014), pp. 50–67. DOI: 10.1016/j.ins.2013.12.011.

[42] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *CoRR* abs/1309.0238 (2013). arXiv: 1309.0238.

[43] Christopher J. C. Burges; Robert Ragno; Quoc Viet Le. "Learning to Rank with Nonsmooth Cost Functions". In: *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*. 2006, pp. 193–200.

[44] Christopher J. C. Burges et al. "Learning to Rank Using Gradient Descent". In: *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*. 2005, pp. 89–96. DOI: 10.1145/1102351.1102363.

[45] Han Cai et al. "Efficient Architecture Search by Network Transformation". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith; Kilian Q. Weinberger. AAAI Press, 2018, pp. 2787–2794.

[46] Fatih Çakir et al. "Deep Metric Learning to Rank". In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 1861–1870. DOI: 10.1109/CVPR.2019.00196.

[47] Zhe Cao et al. "Learning to rank: from pairwise approach to listwise approach". In: *Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007*. Ed. by Zoubin Ghahramani. Vol. 227. ACM International Conference Proceeding Series. ACM, 2007, pp. 129–136. DOI: 10.1145/1273496.1273513.

[48] Rich Caruana; Alexandru Niculescu-Mizil. "An Empirical Comparison of Supervised Learning Algorithms". In: *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*. Ed. by William W. Cohen; Andrew W. Moore. Vol. 148. ACM International Conference Proceeding Series. ACM, 2006, pp. 161–168. DOI: 10.1145/1143844.1143865.

[49] Anup Chalamalla; Ihab F. Ilyas; Mourad Ouzzani; Paolo Papotti. "Descriptive and Prescriptive Data Cleaning". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Curtis E. Dyreson; Feifei Li; M. Tamer Özsu. ACM, 2014, pp. 445–456. DOI: 10.1145/2588555.2610520.

[50] Pete Chapman et al. "The CRISP-DM user guide". In: *4th CRISP-DM SIG Workshop in Brussels in March*. Vol. 1999. sn. 1999.

[51] Weiwei Cheng. "Label Ranking with Probabilistic Models". PhD thesis. University of Marburg, 2012.

[52] François Chollet et al. "Keras: The python deep learning library". In: *Astrophysics source code library* (2018), ascl–1806.

[53] Xiangxiang Chu; Bo Zhang; Ruijun Xu. "Multi-Objective Reinforced Evolution in Mobile Neural Architecture Search". In: *Computer Vision - ECCV 2020 Workshops - Glasgow, UK, August 23-28, 2020, Proceedings, Part IV*. Ed. by Adrien Bartoli; Andrea Fusiello. Vol. 12538. Lecture Notes in Computer Science. Springer, 2020, pp. 99–113. DOI: 10.1007/978-3-030-66823-5_6.

[54] Xu Chu; Ihab F. Ilyas; Sanjay Krishnan; Jiannan Wang. "Data Cleaning: Overview and Emerging Challenges". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan; Georgia Koutrika; Sam Madden. ACM, 2016, pp. 2201–2206. DOI: 10.1145/2882903.2912574.

[55] Xu Chu; Ihab F. Ilyas; Paolo Papotti. "Holistic Data Cleaning: Putting Violations into Context". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. Ed. by Christian S. Jensen; Christopher M. Jermaine; Xiaofang Zhou. IEEE Computer Society, 2013, pp. 458–469. DOI: 10.1109/ICDE.2013.6544847.

[56] R. Collobert; K. Kavukcuoglu; C. Farabet. "Torch7: A Matlab-like Environment for Machine Learning". In: *BigLearn, NIPS Workshop*. 2011.

[57] Corinna Cortes; Vladimir Vapnik. "Support-Vector Networks". In: *Mach. Learn.* 20.3 (1995), pp. 273–297. DOI: 10.1007/BF00994018.

[58] Koby Crammer; Yoram Singer. "Pranking with Ranking". In: *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*. 2001, pp. 641–647.

[59] Bill Curtis et al. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics". In: *IEEE Transactions on software engineering* 2 (1979), pp. 96–104.

[60] Filip Dabek; Jesus J Caban. "A Grammar-Based Approach for Modeling User Interactions and Generating Suggestions during the Data Exploration Process". In: *IEEE transactions on visualization and computer graphics* 23.1 (2016), pp. 41–50.

[61] A Philip Dawid. "Present Position and Potential Developments: Some Personal Views Statistical Theory the Prequential Approach". In: *Journal of the Royal Statistical Society: Series A (General)* 147.2 (1984), pp. 278–290.

[62] Jia Deng et al. "Imagenet: A Large-Scale Hierarchical Image Database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Ieee. 2009, pp. 248–255.

[63] Ramón Díaz-Uriarte; Sara Alvarez de Andrés. "Gene selection and classification of microarray data using random forest". In: *BMC Bioinform.* 7 (2006), p. 3. DOI: 10.1186/1471-2105-7-3.

[64] Tobias Domhan; Jost Tobias Springenberg; Frank Hutter. "Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Qiang Yang; Michael J. Wooldridge. AAAI Press, 2015, pp. 3460–3468.

[65] Pedro M. Domingos; Geoff Hulten. "Mining high-speed data streams". In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000*. Ed. by Raghu Ramakrishnan; Salvatore J. Stolfo; Roberto J. Bayardo; Ismail Parsa. ACM, 2000, pp. 71–80. DOI: 10.1145/347090.347107.

[66] Jin-Dong Dong et al. “DPP-Net: Device-Aware Progressive Search for Pareto-Optimal Neural Architectures”. In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XI*. Ed. by Vittorio Ferrari; Martial Hebert; Cristian Sminchisescu; Yair Weiss. Vol. 11215. Lecture Notes in Computer Science. Springer, 2018, pp. 540–555. DOI: 10.1007/978-3-030-01252-6_32.

[67] Xuanyi Dong; Lu Liu; Katarzyna Musial; Bogdan Gabrys. “NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (2021). DOI: 10.1109/TPAMI.2021.3054824.

[68] Xuanyi Dong; Yi Yang. “NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[69] Hua Duan et al. “An incremental learning algorithm for Lagrangian support vector machines”. In: *Pattern Recognit. Lett.* 30.15 (2009), pp. 1384–1391. DOI: 10.1016/j.patrec.2009.07.006.

[70] John C. Duchi; Elad Hazan; Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *J. Mach. Learn. Res.* 12 (2011), pp. 2121–2159.

[71] Russell C Eberhart; Yuhui Shi. “Comparison between Genetic Algorithms and Particle Swarm Optimization”. In: *International Conference on Evolutionary Programming*. Springer. 1998, pp. 611–616.

[72] Henry R. Ehrenberg et al. “Data programming with DDLite: putting humans in a different part of the loop”. In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Carsten Binnig; Alan D. Fekete; Arnab Nandi. ACM, 2016, p. 13. DOI: 10.1145/2939502.2939515.

[73] Thomas Elsken; Jan Hendrik Metzen; Frank Hutter. “Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. 2019.

[74] Thomas Elsken; Jan Hendrik Metzen; Frank Hutter. “Neural Architecture Search: A Survey”. In: *J. Mach. Learn. Res.* 20 (2019), 55:1–55:21.

[75] Thomas Elsken; Jan Hendrik Metzen; Frank Hutter. “Simple and efficient architecture search for Convolutional Neural Networks”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018.

[76] Alex Endert; Patrick Fiaux; Chris North. “Semantic Interaction for Sensemaking: Inferring Analytical Reasoning for Model Steering”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2879–2888.

[77] Tolga Ergen; Suleyman Serdar Kozat. “Efficient Online Learning Algorithms Based on LSTM Neural Networks”. In: *IEEE transactions on neural networks and learning systems* 29.8 (2017), pp. 3772–3783.

[78] Stefan Falkner; Aaron Klein; Frank Hutter. “BOHB: Robust and Efficient Hyperparameter Optimization at Scale”. In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy; Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1436–1445.

[79] Usama M. Fayyad; Gregory Piatetsky-Shapiro; Padhraic Smyth. “Knowledge Discovery and Data Mining: Towards a Unifying Framework”. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*. Ed. by Evangelos Simoudis; Jiawei Han; Usama M. Fayyad. AAAI Press, 1996, pp. 82–88.

[80] Matthias Feurer et al. “Auto-Sklearn 2.0: The next Generation”. In: *arXiv* abs/2007.04074 (2020). arXiv: 2007.04074.

[81] Matthias Feurer et al. “Efficient and Robust Automated Machine Learning”. In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. 2015, pp. 2962–2970.

[82] E Fiesler. “Neural Network Classification and Formalization”. In: *Computer Standards & Interfaces* 16.3 (July 1994), pp. 231–239. ISSN: 09205489. DOI: 10.1016/0920-5489(94)90014-0.

[83] Evelyn Fix; Joseph Lawson Hodges. “Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties”. In: *International Statistical Review/Revue Internationale de Statistique* 57.3 (1989), pp. 238–247.

[84] Dario Floreano; Peter Dürr; Claudio Mattiussi. “Neuroevolution: From Architectures to Learning”. In: *Evolutionary intelligence* 1.1 (2008), pp. 47–62.

[85] Félix-Antoine Fortin et al. “DEAP: evolutionary algorithms made easy”. In: *J. Mach. Learn. Res.* 13 (2012), pp. 2171–2175.

[86] William J Frawley; Gregory Piatetsky-Shapiro; Christopher J Matheus. “Knowledge Discovery in Databases: An Overview”. In: *AI magazine* 13.3 (1992), pp. 57–57.

[87] Yoav Freund; Robert E Schapire. “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”. In: *Journal of Computer and System Sciences* 55.1 (Aug. 1997), pp. 119–139. ISSN: 00220000. DOI: 10.1006/jcss.1997.1504.

[88] Jerome H Friedman. “Stochastic Gradient Boosting”. In: *Computational statistics & data analysis* 38.4 (2002), pp. 367–378.

[89] Jerome H. Friedman; Jacqueline J. Meulman. “Multiple Additive Regression Trees with Application in Epidemiology”. In: *Statistics in Medicine* 22.9 (2003), pp. 1365–1381. DOI: 10.1002/sim.1501. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sim.1501.

[90] Jörg Frochte. *Maschinelles Lernen: Grundlagen Und Algorithmen in Python*. Carl Hanser Verlag GmbH Co KG, 2020.

[91] João Gama; Pedro Medas; Gladys Castillo; Pedro Pereira Rodrigues. “Learning with Drift Detection”. In: *Advances in Artificial Intelligence - SBIA 2004, 17th Brazilian Symposium on Artificial Intelligence, São Luis, Maranhão, Brazil, September 29 - October 1, 2004, Proceedings*. Ed. by Ana L. C. Bazzan; Sofiane Labidi. Vol. 3171. Lecture Notes in Computer Science. Springer, 2004, pp. 286–295. DOI: 10.1007/978-3-540-28645-5_29.

[92] João Gama et al. “A survey on concept drift adaptation”. In: *ACM Comput. Surv.* 46.4 (2014), 44:1–44:37. DOI: 10.1145/2523813.

[93] Steven Gardner et al. “Constrained Multi-Objective Optimization for Automated Machine Learning”. In: *2019 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2019, pp. 364–373.

[94] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. " O’Reilly Media, Inc.", 2019.

[95] Yolanda Gil et al. “Towards Human-Guided Machine Learning”. In: *Proceedings of the 24th International Conference on Intelligent User Interfaces, IUI 2019, Marina Del Ray, CA, USA, March 17-20, 2019*. 2019, pp. 614–624. DOI: 10.1145/3301275.3302324.

[96] Chaitanya Gokhale et al. “Corleone: Hands-off Crowdsourcing for Entity Matching”. In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Curtis E. Dyreson; Feifei Li; M. Tamer Özsu. ACM, 2014, pp. 601–612. DOI: 10.1145/2588555.2588576.

[97] Jacob Goldberger; Sam T. Roweis; Geoffrey E. Hinton; Ruslan Salakhutdinov. “Neighbourhood Components Analysis”. In: *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, NIPS 2004, December 13-18, 2004, Vancouver, British Columbia, Canada]*. 2004, pp. 513–520.

[98] Heitor M Gomes et al. “Adaptive random forests for evolving data stream classification”. In: *Machine Learning* 106.9 (2017). Publisher: Springer, pp. 1469–1495.

[99] Heitor Murilo Gomes; Jesse Read; Albert Bifet. “Streaming Random Patches for Evolving Data Stream Classification”. In: *2019 IEEE International Conference on Data Mining (ICDM)*. Beijing, China: IEEE, Nov. 2019, pp. 240–249. ISBN: 978-1-72814-604-1. DOI: 10.1109/ICDM.2019.00034.

[100] Ana González; José R Dorronsoro. “Natural Conjugate Gradient Training of Multilayer Perceptrons”. In: *Neurocomputing* 71.13-15 (2008), pp. 2499–2506.

[101] Ian Goodfellow; Yoshua Bengio; Aaron Courville. *Deep Learning*. MIT Press, 2016.

[102] Thore Graepel; Ralf Herbrich; Klaus Obermayer. “Bayesian Transduction”. In: *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*. Ed. by Sara A. Solla; Todd K. Leen; Klaus-Robert Müller. The MIT Press, 1999, pp. 456–462.

[103] Isabelle Guyon; Jason Weston; Stephen Barnhill; Vladimir Vapnik. “Gene Selection for Cancer Classification Using Support Vector Machines”. In: *Machine learning* 46.1 (2002), pp. 389–422.

[104] Mark Hall et al. “The WEKA Data Mining Software: An Update”. In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[105] Mark A. Hall; Lloyd A. Smith. “Feature Selection for Machine Learning: Comparing a Correlation-Based Filter Approach to the Wrapper”. In: *Proceedings of the Twelfth International Florida Artificial Intelligence Research Society Conference, May 1-5, 1999, Orlando, Florida, USA*. Ed. by Amruth N. Kumar; Ingrid Russell. AAAI Press, 1999, pp. 235–239.

[106] Nikolaus Hansen. “The CMA Evolution Strategy: A Comparing Review”. In: *Towards a New Evolutionary Computation - Advances in the Estimation of Distribution Algorithms*. Ed. by José Antonio Lozano; Pedro Larrañaga; Iñaki Inza; Endika Bengoetxea. Vol. 192. Studies in Fuzziness and Soft Computing. Springer, 2006, pp. 75–102. DOI: 10.1007/3-540-32494-1_4.

[107] Kaiming He; Xiangyu Zhang; Shaoqing Ren; Jian Sun. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[108] Florian Heimerl; Steffen Koch; Harald Bosch; Thomas Ertl. “Visual Classifier Training for Text Document Retrieval”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2839–2848.

[109] Geoffrey Hinton et al. “Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups”. In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[110] Geoffrey E Hinton; Simon Osindero; Yee-Whye Teh. "A Fast Learning Algorithm for Deep Belief Nets". In: *Neural computation* 18.7 (2006), pp. 1527–1554.

[111] Sepp Hochreiter; Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[112] Steven CH Hoi; Doyen Sahoo; Jing Lu; Peilin Zhao. "Online learning: A comprehensive survey". In: *Neurocomputing* 459 (2021). Publisher: Elsevier, pp. 249–289.

[113] John H Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT press, 1992.

[114] Chi-Hung Hsu et al. "MONAS: Multi-objective Neural Architecture Search Using Reinforcement Learning". In: *CoRR* abs/1806.10332 (2018). arXiv: 1806.10332.

[115] Gao Huang; Zhuang Liu; Laurens Van Der Maaten; Kilian Q Weinberger. "Densely Connected Convolutional Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 4700–4708.

[116] Geoff Hulten; Laurie Spencer; Pedro Domingos. "Mining time-changing data streams". In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. 2001, pp. 97–106.

[117] Frank Hutter; Holger H. Hoos; Kevin Leyton-Brown. "Sequential Model-Based Optimization for General Algorithm Configuration". In: *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*. 2011, pp. 507–523. DOI: 10.1007/978-3-642-25566-3_40.

[118] Frank Hutter; Lars Kotthoff; Joaquin Vanschoren, eds. *Automated Machine Learning: Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-05317-8 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5.

[119] Hutter, Frank and Lücke, Jörg and Schmidt-Thieme, Lars. "Beyond Manual Tuning of Hyperparameters". In: *KI-Künstliche Intelligenz* 29.4 (2015), pp. 329–337.

[120] Osman Ali Sadek Ibrahim; Dario Landa-Silva. "ES-Rank: Evolution Strategy Learning to Rank Approach". In: *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*. 2017, pp. 944–950. DOI: 10.1145/3019612.3019696.

[121] Alexandru-Ionut Imbrea. "Automated Machine Learning Techniques for Data Streams". In: *CoRR* abs/2106.07317 (2021). arXiv: 2106.07317.

[122] Yesmina Jaâfra; Jean Luc Laurent; Aline Deruyver; Mohamed Saber Naceur. "Robust Reinforcement Learning for Autonomous Driving". In: *Deep Reinforcement Learning Meets Structured Prediction, ICLR 2019 Workshop, New Orleans, Louisiana, United States, May 6, 2019*. OpenReview.net, 2019.

[123] Lakhmi C. Jain; Manjeevan Seera; Chee Peng Lim; P. Balasubramaniam. "A Review of Online Learning in Supervised Neural Networks". In: *Neural Computing and Applications* 25.3-4 (Sept. 2014), pp. 491–509. ISSN: 0941-0643, 1433-3058. DOI: 10.1007/s00521-013-1534-4.

[124] Liangxiao Jiang; Chaoqun Li. "Scaling Up the Accuracy of Decision-Tree Classifiers: A Naive-Bayes Combination". In: *J. Comput.* 6.7 (2011), pp. 1325–1331. DOI: 10.4304/jcp.6.7.1325-1331.

[125] Haifeng Jin; Qingquan Song; Xia Hu. "Auto-Keras: An Efficient Neural Architecture Search System". In: *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. 2019, pp. 1946–1956. DOI: 10.1145/3292500.3330648.

[126] Donald R. Jones; Matthias Schonlau; William J. Welch. “Efficient Global Optimization of Expensive Black-Box Functions”. In: *J. Glob. Optim.* 13.4 (1998), pp. 455–492. DOI: 10.1023/A:1008306431147.

[127] Ambika Kaul; Saket Maheshwary; Vikram Pudi. “AutoLearn — Automated Feature Generation and Selection”. In: *2017 IEEE International Conference on Data Mining (ICDM)*. New Orleans, LA: IEEE, Nov. 2017, pp. 217–226. ISBN: 978-1-5386-3835-4. DOI: 10.1109/ICDM.2017.31.

[128] Ye-Hoon Kim; Bhargava Reddy; Sojung Yun; Chanwon Seo. “Nemo: Neuro-evolution with Multi-objective Optimization of Deep Neural Network for Speed and Accuracy”. In: *JMLR: Workshop and Conference Proceedings*. Vol. 1. 2017, pp. 1–8.

[129] Diederik P. Kingma; Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio; Yann LeCun. 2015.

[130] Nikita Klyuchnikov et al. “NAS-Bench-NLP: Neural Architecture Search Benchmark for Natural Language Processing”. In: *CoRR* abs/2006.07116 (2020). arXiv: 2006.07116.

[131] Y. Kodratoff. “From machine learning towards knowledge discovery in databases”. English. In: *IET Conference Proceedings* (1995), 5–5(1).

[132] Daphne Koller; Mehran Sahami. “Toward Optimal Feature Selection”. In: *Machine Learning, Proceedings of the Thirteenth International Conference (ICML ’96), Bari, Italy, July 3-6, 1996*. Ed. by Lorenza Saitta. Morgan Kaufmann, 1996, pp. 284–292.

[133] J Zico Kolter; Marcus A Maloof. “Dynamic Weighted Majority: An Ensemble Method for Drifting Concepts”. In: *The Journal of Machine Learning Research* 8 (2007), pp. 2755–2790.

[134] Brent Komer; James Bergstra; Chris Eliasmith. “Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn”. In: *ICML Workshop on AutoML*. Vol. 9. Citeseer. 2014, p. 50.

[135] Lars Kotthoff et al. “Auto-WEKA 2.0: Automatic Model Selection and Hyperparameter Optimization in WEKA”. In: *Journal of Machine Learning Research* 18 (2017), 25:1–25:5.

[136] Yuki Koyama; Daisuke Sakamoto; Takeo Igarashi. “Selph: Progressive Learning and Support of Manual Photo Color Enhancement”. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 2016, pp. 2520–2532.

[137] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[138] Alex Krizhevsky; Ilya Sutskever; Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Communications of the ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3065386.

[139] Cedric Kulbach; Patrick Philipp; Steffen Thoma. “Personalized Automated Machine Learning”. In: *Santiago de Compostela* (2020), p. 8.

[140] Cedric Kulbach; Steffen Thoma. “Personalized Neural Architecture Search”. In: *2021 International Conference on Data Mining Workshops (ICDMW)*. IEEE. 2021, pp. 581–590.

[141] Cedric Kulbach et al. “Evolution-Based Online Automated Machine Learning”. In: *26th Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Chengdu, China, 2022.

[142] Brian Kulis. “Metric Learning: A Survey”. In: *Foundations and Trends® in Machine Learning* 5.4 (2013), pp. 287–364. ISSN: 1935-8237, 1935-8245. DOI: 10.1561/2200000019.

[143] Hugo Larochelle et al. “An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation”. In: *Proceedings of the 24th International Conference on Machine Learning*. 2007, pp. 473–480.

[144] Trang T Le; Weixuan Fu; Jason H Moore. "Scaling Tree-Based Automated Machine Learning to Biomedical Big Data with a Feature Set Selector". In: *Bioinformatics (Oxford, England)* 36.1 (2020), pp. 250–256.

[145] Yann LeCun; Léon Bottou; Yoshua Bengio; Patrick Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[146] Yann LeCun et al. "Generalization and Network Design Strategies". In: *Connectionism in perspective* 19.143-155 (1989), p. 18.

[147] Erin LeDell; Sebastien Poirier. "H2O AutoML: Scalable Automatic Machine Learning". In: *7th ICML Workshop on Automated Machine Learning (AutoML)* (July 2020).

[148] E.W.M. Lee; Chee Peng Lim; R.K.K. Yuen; S.M. Lo. "A Hybrid Neural Network Model for Noisy Data Regression". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 34.2 (2004), pp. 951–960. DOI: 10.1109/TSMCB.2003.818440.

[149] Stefan Leijnen; Fjodor van Veen. "The Neural Network Zoo". In: *Multidisciplinary Digital Publishing Institute Proceedings*. Vol. 47. 2020, p. 9.

[150] Dewei Li; Yingjie Tian. "Survey and Experimental Study on Metric Learning Methods". In: *Neural Networks* 105 (Sept. 2018), pp. 447–462. ISSN: 08936080. DOI: 10.1016/j.neunet.2018.06.003.

[151] Ping Li; Christopher J. C. Burges; Qiang Wu. "McRank: Learning to Rank Using Multiple Classification and Gradient Boosting". In: *Advances in Neural Information Processing Systems 20, Proceedings of the Twenty-First Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 3-6, 2007*. 2007, pp. 897–904.

[152] Tao Li; Chengliang Zhang; Mitsunori Ogihara. "A Comparative Study of Feature Selection and Multiclass Classification Methods for Tissue Classification Based on Gene Expression". In: *Bioinform.* 20.15 (2004), pp. 2429–2437. DOI: 10.1093/bioinformatics/bth267.

[153] Hanxiao Liu; Karen Simonyan; Yiming Yang. "DARTS: Differentiable Architecture Search". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[154] Tong Liu et al. "Fast Adaptive Gradient RBF Networks for Online Learning of Nonstationary Time Series". In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 2015–2030.

[155] Yuqiao Liu et al. "A Survey on Evolutionary Neural Architecture Search". In: *IEEE Transactions on Neural Networks and Learning Systems* (2021), pp. 1–21. ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2021.3100554.

[156] Jesus L Lobo; Javier Del Ser; Albert Bifet; Nikola Kasabov. "Spiking Neural Networks and Online Learning: An Overview and Perspectives". In: *Neural Networks* 121 (2020), pp. 88–100.

[157] Jesus L. Lobo et al. "Evolving Spiking Neural Networks for Online Learning over Drifting Data Streams". In: *Neural Networks* 108 (Dec. 2018), pp. 1–19. ISSN: 08936080. DOI: 10.1016/j.neunet.2018.07.014.

[158] Ilya Loshchilov; Frank Hutter. "Decoupled Weight Decay Regularization". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[159] Zhichao Lu et al. "NSGA-Net: Neural Architecture Search Using Multi-Objective Genetic Algorithm". In: *Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*. Ed. by Anne Auger; Thomas Stützle. ACM, 2019, pp. 419–427. DOI: 10.1145/3321707.3321729.

[160] Sean Luke. *Essentials of Metaheuristics: A Set of Undergraduate Lecture Notes; Online Version 2.0*. 2. ed. S.l.: Lulu, 2013. ISBN: 978-1-300-54962-8.

[161] Gang Luo. "A Review of Automatic Selection Methods for Machine Learning Algorithms and Hyper-Parameter Values". In: *Network Modeling Analysis in Health Informatics and Bioinformatics* 5.1 (2016), pp. 1–16.

[162] Warren S McCulloch; Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[163] Abhinav Mehrotra et al. "Nas-Bench-Asr: Reproducible Neural Architecture Search for Speech Recognition". In: *International Conference on Learning Representations*. 2020.

[164] Yash Mehta et al. "NAS-Bench-Suite: NAS Evaluation Is (Now) Surprisingly Easy". In: *International Conference on Learning Representations*. 2022.

[165] Manuel Mejía-Lavalle; Enrique Sucar; Gustavo Arroyo. "Feature Selection with a Perceptron Neural Net". In: *Proceedings of the International Workshop on Feature Selection for Data Mining*. 2006, pp. 131–135.

[166] Gábor Melis; Chris Dyer; Phil Blunsom. "On the State of the Art of Evaluation in Neural Language Models". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[167] Hector Mendoza et al. "Towards Automatically-Tuned Neural Networks". In: *Workshop on Automatic Machine Learning*. PMLR. 2016, pp. 58–65.

[168] George A Miller. "WordNet: A Lexical Database for English". In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

[169] Leandro L. Minku; Xin Yao. "DDD: A New Ensemble Approach for Dealing with Concept Drift". In: *IEEE Transactions on Knowledge and Data Engineering* 24.4 (Apr. 2012), pp. 619–633. ISSN: 1041-4347. DOI: 10.1109/TKDE.2011.58.

[170] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Series in Computer Science. New York: McGraw-Hill, 1997. ISBN: 978-0-07-042807-2.

[171] Douglas C. Montgomery. *Design and Analysis of Experiments*. Eighth edition. Hoboken, NJ: John Wiley & Sons, Inc, 2013. ISBN: 978-1-118-14692-7.

[172] Jacob Montiel; Jesse Read; Albert Bifet; Talel Abdessalem. "Scikit-Multiflow: A Multi-Output Streaming Framework". In: *The Journal of Machine Learning Research* 19.1 (2018), pp. 2915–2914.

[173] Jacob Montiel et al. "River: machine learning for streaming data in Python". In: *J. Mach. Learn. Res.* 22 (2021), 110:1–110:8.

[174] Michael A Nielsen. *Neural Networks and Deep Learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[175] Yohei Nose; Akira Kojima; Hideyuki Kawabata; Tetsuo Hironaka. "A Study on a Lane Keeping System Using CNN for Online Learning of Steering Control from Real Time Images". In: *2019 34th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*. JeJu, Korea (South): IEEE, June 2019, pp. 1–4. ISBN: 978-1-72813-271-6. DOI: 10.1109/ITC-CSCC.2019.8793348.

[176] Ivo F. D. Oliveira; Nir Ailon; Ori Davidov. "A New and Flexible Approach to the Analysis of Paired Comparison Data". In: *Journal of Machine Learning Research* 19 (2018), 60:1–60:29.

[177] Randal S. Olson; Nathan Bartley; Ryan J. Urbanowicz; Jason H. Moore. "Evaluation of a Tree-Based Pipeline Optimization Tool for Automating Data Science". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. New York, NY, USA: ACM, 2016, pp. 485–492. ISBN: 978-1-4503-4206-3. DOI: 10.1145/2908812.2908918.

[178] Tom O'Malley et al. *KerasTuner*. https://github.com/keras-team/keras-tuner. 2019.

[179] Chia Huey Ooi; Patrick Tan. "Genetic Algorithms Applied to Multi-Class Prediction for the Analysis of Gene Expression Data". In: *Bioinform.* 19.1 (2003), pp. 37–44. DOI: 10.1093/bioinformatics/19.1.37.

[180] Nikunj C Oza; Stuart Russell. "Experimental Comparisons of Online and Batch Versions of Bagging and Boosting". In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2001, pp. 359–364.

[181] Nikunj C Oza; Stuart J Russell. "Online bagging and boosting". In: *International workshop on artificial intelligence and statistics*. tex.organization: PMLR. 2001, pp. 229–236.

[182] Lawrence Page. "Method for Node Ranking in a Linked Database". In: *US Patent* 6,285,999 (2001).

[183] Liang Pang et al. "SetRank: Learning a Permutation-Invariant Ranking Model for Information Retrieval". In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 499–508. ISBN: 978-1-4503-8016-4.

[184] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035.

[185] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830.

[186] Beatriz Pérez-Sánchez; Oscar Fontenla-Romero; Bertha Guijarro-Berdiñas. "A Review of Adaptive Online Learning for Artificial Neural Networks". In: *Artificial Intelligence Review* 49.2 (2018), pp. 281–299.

[187] Florian Pfisterer; Stefan Coors; Janek Thomas; Bernd Bischl. "Multi-Objective Automatic Machine Learning with AutoxgboostMC". In: *CoRR* abs/1908.10796 (2019). arXiv: 1908.10796.

[188] Hieu Pham et al. "Efficient Neural Architecture Search via Parameter Sharing". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*. 2018, pp. 4092–4101.

[189] Patrick Raoul Philipp. "Decision-Making with Multi-Step Expert Advice on the Web". PhD thesis. Karlsruher Institut für Technologie (KIT), 2019. 218 pp. DOI: 10.5445/IR/1000093522.

[190] Gregory Piatetsky-Shapiro. "Knowledge Discovery in Real Databases: A Report on the IJCAI-89 Workshop". In: *AI magazine* 11.4 (1990), pp. 68–68.

[191] Guo-Jun Qi et al. "An Efficient Sparse Metric Learning in High-Dimensional Space via $l_{(1)}$-Penalized Log-Determinant Regularization". In: *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*. Ed. by Andrea Pohoreckyj Danyluk; Léon Bottou; Michael L. Littman. Vol. 382. ACM International Conference Proceeding Series. ACM, 2009, pp. 841–848. DOI: 10.1145/1553374.1553482.

[192] J. R. Quinlan. "Induction of Decision Trees". In: *Machine Learning* 1.1 (Mar. 1986), pp. 81–106. ISSN: 0885-6125, 1573-0565. DOI: 10.1007/BF00116251.

[193] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1-55860-238-0.

[194] John Ross Quinlan; Paul J Compton; KA Horn; Leslie Lazarus. "Inductive Knowledge Acquisition: A Case Study". In: *Proceedings of the Second Australian Conference on Applications of Expert Systems*. 1987, pp. 137–156.

[195] Erhard Rahm; Hong Hai Do. "Data Cleaning: Problems and Current Approaches". In: *IEEE Data Eng. Bull.* 23.4 (2000), pp. 3–13.

[196] Jesse Read; Peter Reutemann; Bernhard Pfahringer; Geoff Holmes. "MEKA: A Multi-Label/Multi-Target Extension to Weka". In: *Journal of Machine Learning Research* 17.21 (2016), pp. 1–5.

[197] Esteban Real; Alok Aggarwal; Yanping Huang; Quoc V Le. "Regularized Evolution for Image Classifier Architecture Search". In: *Proceedings of the Aaai Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 4780–4789.

[198] Esteban Real; Sherry Moore; Andrew Selle et. al. "Large-Scale Evolution of Image Classifiers". In: *ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 2017, pp. 2902–2911.

[199] Jason D. M. Rennie; Lawrence Shih; Jaime Teevan; David R. Karger. "Tackling the Poor Assumptions of Naive Bayes Text Classifiers". In: *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*. Ed. by Tom Fawcett; Nina Mishra. AAAI Press, 2003, pp. 616–623.

[200] Roman Rosipal; Mark Girolami; Leonard J Trejo; Andrzej Cichocki. "Kernel PCA for Feature Extraction and De-Noising in Nonlinear Regression". In: *Neural Computing & Applications* 10.3 (2001), pp. 231–243.

[201] David E Rumelhart; Geoffrey E Hinton; Ronald J Williams. "Learning Representations by Back-Propagating Errors". In: *nature* 323.6088 (1986), pp. 533–536.

[202] Stuart J. Russell; Peter Norvig; Ernest Davis. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall Series in Artificial Intelligence. Upper Saddle River: Prentice Hall, 2010. ISBN: 978-0-13-604259-4.

[203] Y. Saeys; I. Inza; P. Larranaga. "A Review of Feature Selection Techniques in Bioinformatics". In: *Bioinformatics* 23.19 (Oct. 2007), pp. 2507–2517. ISSN: 1367-4803, 1460-2059. DOI: 10.1093/bioinformatics/btm344.

[204] Doyen Sahoo; Quang Pham; Jing Lu; Steven C. H. Hoi. "Online Deep Learning: Learning Deep Neural Networks on the Fly". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. Stockholm, Sweden: International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 2660–2666. ISBN: 978-0-9992411-2-7. DOI: 10.24963/ijcai.2018/369.

[205] Cristiano Saltori; Subhankar Roy; Nicu Sebe; Giovanni Iacca. "Regularized Evolutionary Algorithm for Dynamic Neural Topology Search". In: *Image Analysis and Processing - ICIAP 2019 - 20th International Conference, Trento, Italy, September 9-13, 2019, Proceedings, Part I*. Ed. by Elisa Ricci et al. Vol. 11751. Lecture Notes in Computer Science. Springer, 2019, pp. 219–230. DOI: 10.1007/978-3-030-30642-7_20.

[206] Franco Scarselli et al. "The Graph Neural Network Model". In: *IEEE Trans. Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[207] D. Sculley. "Combined Regression and Ranking". In: *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*. 2010, pp. 979–988. DOI: 10.1145/1835804.1835928.

[208] Ramprasaath R. Selvaraju et al. “Grad-Cam: Visual Explanations from Deep Networks via Gradient-Based Localization”. In: *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 2017, pp. 618–626. DOI: 10.1109/ICCV.2017.74.

[209] Bobak Shahriari et al. “Taking the Human Out of the Loop: A Review of Bayesian Optimization”. In: *Proc. IEEE* 104.1 (2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

[210] Sheng Wan; L.E. Banta. “Parameter Incremental Learning Algorithm for Neural Networks”. In: *IEEE Transactions on Neural Networks* 17.6 (Nov. 2006), pp. 1424–1438. ISSN: 1045-9227, 1941-0093. DOI: 10.1109/TNN.2006.880581.

[211] J Sayyad Shirabad; Tim J Menzies, et al. “The PROMISE Repository of Software Engineering Databases”. In: *School of Information Technology and Engineering, University of Ottawa, Canada* 24 (2005).

[212] Wojciech Siedlecki; Jack Sklansky. “On Automatic Feature Selection”. In: *Handbook of Pattern Recognition & Computer Vision*. USA: World Scientific Publishing Co., Inc., 1993, pp. 63–87. ISBN: 981-02-1136-8.

[213] Julien Siems et al. “NAS-Bench-301 and the Case for Surrogate Benchmarks for Neural Architecture Search”. In: *CoRR* abs/2008.09777 (2020). arXiv: 2008.09777.

[214] D. Simon. *Evolutionary Optimization Algorithms*. Wiley, 2013. ISBN: 9781118659502.

[215] Karen Simonyan; Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio; Yann LeCun. 2015.

[216] David B. Skalak. “Prototype and Feature Selection by Sampling and Random Mutation Hill Climbing Algorithms”. In: *Machine Learning, Proceedings of the Eleventh International Conference, Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994*. Ed. by William W. Cohen; Haym Hirsh. Morgan Kaufmann, 1994, pp. 293–301. DOI: 10.1016/b978-1-55860-335-6.50043-x.

[217] Jasper Snoek; Hugo Larochelle; Ryan P Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Advances in neural information processing systems* 25 (2012).

[218] Jasper Snoek et al. “Scalable Bayesian Optimization Using Deep Neural Networks”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach; David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 2171–2180.

[219] W Nick Street; YongSeog Kim. “A streaming ensemble algorithm (SEA) for large-scale classification”. In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. 2001, pp. 377–382.

[220] Masanori Suganuma; Shinichi Shirakawa; Tomoharu Nagao. “A Genetic Programming Approach to Designing Convolutional Neural Network Architectures”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2017, pp. 497–504.

[221] Yanan Sun; Bing Xue; Mengjie Zhang; Gary G Yen. “Completely Automated CNN Architecture Design Based on Blocks”. In: *IEEE transactions on neural networks and learning systems* 31.4 (2019), pp. 1242–1254.

[222] Yanan Sun et al. “Automatically Designing CNN Architectures Using the Genetic Algorithm for Image Classification”. In: *IEEE transactions on cybernetics* 50.9 (2020), pp. 3840–3854.

[223] Thomas Swearingen et al. “ATM: A Distributed, Collaborative, Scalable System for Automated Machine Learning”. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 151–162.

[224] Nadeem Ahmed Syed; Huan Liu; Kah Kay Sung. "Handling Concept Drifts in Incremental Learning with Support Vector Machines". In: *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 15-18, 1999*. Ed. by Usama M. Fayyad; Surajit Chaudhuri; David Madigan. ACM, 1999, pp. 317–321. DOI: 10.1145/312129.312267.

[225] Christian Szegedy et al. "Going Deeper with Convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[226] Mingxing Tan et al. "MnasNet: Platform-aware Neural Architecture Search for Mobile". In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 2820–2828. DOI: 10.1109/CVPR.2019.00293.

[227] Janek Thomas; Stefan Coors; Bernd Bischl. "Automatic Gradient Boosting". In: *International Workshop on Automatic Machine Learning at ICML*. 2018.

[228] Chris Thornton; Frank Hutter; Holger H. Hoos; Kevin Leyton-Brown. "Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms". In: *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. 2013, pp. 847–855. DOI: 10.1145/2487575.2487629.

[229] Haiman Tian; Shu-Ching Chen; Mei-Ling Shyu; Stuart Rubin. "Automated Neural Network Construction with Similarity Sensitive Evolutionary Algorithms". In: *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE. 2019, pp. 283–290.

[230] Tin Kam Ho. "Random Decision Forests". In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. Vol. 1. Montreal, Que., Canada: IEEE Comput. Soc. Press, 1995, pp. 278–282. ISBN: 978-0-8186-7128-9. DOI: 10.1109/ICDAR.1995.598994.

[231] Eugene Tuv; Alexander Borisov; George Runger; Kari Torkkola. "Feature Selection with Ensembles, Artificial Variables, and Redundancy Elimination". In: *The Journal of Machine Learning Research* 10 (2009), pp. 1341–1366.

[232] Hamed Valizadegan; Rong Jin; Ruofei Zhang; Jianchang Mao. "Learning to Rank by Optimizing Ndcg Measure". In: *Advances in neural information processing systems* 22 (2009).

[233] Jan N. van Rijn; Geoffrey Holmes; Bernhard Pfahringer; Joaquin Vanschoren. "Having a Blast: Meta-Learning and Heterogeneous Ensembles for Data Streams". In: *2015 IEEE International Conference on Data Mining*. Atlantic City, NJ, USA: IEEE, Nov. 2015, pp. 1003–1008. ISBN: 978-1-4673-9504-5. DOI: 10.1109/ICDM.2015.55.

[234] Joaquin Vanschoren; Jan N Van Rijn; Bernd Bischl; Luis Torgo. "OpenML: Networked Science in Machine Learning". In: *ACM SIGKDD Explorations Newsletter* 15.2 (2014), pp. 49–60.

[235] Jiannan Wang; Tim Kraska; Michael J. Franklin; Jianhua Feng. "CrowdER: Crowdsourcing Entity Resolution". In: *Proc. VLDB Endow.* 5.11 (2012), pp. 1483–1494. DOI: 10.14778/2350229.2350263.

[236] Xinying Wang; Min Han. "Online Sequential Extreme Learning Machine with Kernels for Nonstationary Time Series Prediction". In: *Neurocomputing* 145 (Dec. 2014), pp. 90–97. ISSN: 09252312. DOI: 10.1016/j.neucom.2014.05.068.

[237] Kilian Q Weinberger; Lawrence K Saul. "Distance Metric Learning for Large Margin Nearest Neighbor Classification." In: *Journal of machine learning research* 10.2 (2009).

[238] BP Welford. "Note on a Method for Calculating Corrected Sums of Squares and Products". In: *Technometrics : a journal of statistics for the physical, chemical, and engineering sciences* 4.3 (1962), pp. 419–420.

[239] Tino Werner. "A review on instance ranking problems in statistical learning". In: *Mach. Learn.* 111.2 (2022), pp. 415–463. DOI: 10.1007/s10994-021-06122-3.

[240] Colin White; Willie Neiswanger; Yash Savani. "BANANAS: Bayesian Optimization with Neural Architectures for Neural Architecture Search". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 10293–10301.

[241] Darrell Whitley. "An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls". In: *Information and software technology* 43.14 (2001), pp. 817–831.

[242] Gerhard Widmer; Miroslav Kubat. "Learning in the Presence of Concept Drift and Hidden Contexts". In: *Machine learning* 23.1 (1996), pp. 69–101.

[243] David H. Wolpert. "Stacked generalization". In: *Neural Networks* 5.2 (1992), pp. 241–259. DOI: 10.1016/S0893-6080(05)80023-1.

[244] Eugene Wu; Samuel Madden. "Scorpion: Explaining Away Outliers in Aggregate Queries". In: *Proc. VLDB Endow.* 6.8 (2013), pp. 553–564. DOI: 10.14778/2536354.2536356.

[245] Lingxi Xie; Alan L. Yuille. "Genetic CNN". In: *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 2017, pp. 1388–1397. DOI: 10.1109/ICCV.2017.154.

[246] Eric P. Xing; Andrew Y. Ng; Michael I. Jordan; Stuart J. Russell. "Distance Metric Learning with Application to Clustering with Side-Information". In: *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002, December 9-14, 2002, Vancouver, British Columbia, Canada]*. Ed. by Suzanna Becker; Sebastian Thrun; Klaus Obermayer. MIT Press, 2002, pp. 505–512.

[247] Chris Ying et al. "NAS-Bench-101: Towards Reproducible Neural Architecture Search". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri; Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, June 9–15, 2019, pp. 7105–7114.

[248] Lei Yu; Huan Liu. "Efficient Feature Selection via Analysis of Relevance and Redundancy". In: *J. Mach. Learn. Res.* 5 (2004), pp. 1205–1224.

[249] Harry Zhang. "The Optimality of Naive Bayes". In: *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA*. Ed. by Valerie Barr; Zdravko Markov. AAAI Press, 2004, pp. 562–567.

[250] Quanlu Zhang et al. "Retiarii: A Deep Learning Exploratory-Training Framework". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. Nov. 2020.

[251] Shifei Zhou; Kin Keung Lai. "An Improved emd Online Learning-Based Model for Gold Market Forecasting". In: *Intelligent Decision Technologies*. Springer, 2011, pp. 75–84.

[252] Lucas Zimmer; Marius Lindauer; Frank Hutter. "Auto-Pytorch: Multi-Fidelity Metalearning for Efficient and Robust autoDL". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.9 (2021), pp. 3079–3090.

[253] Marc-André Zöller; Marco F. Huber. "Survey on Automated Machine Learning". In: *CoRR* abs/1904.12054 (2019). arXiv: 1904.12054.

[254] D. Zongker; A. Jain. "Algorithms for Feature Selection: An Evaluation". In: *Proceedings of 13th International Conference on Pattern Recognition*. Vol. 2. Aug. 1996, 18–22 vol.2. DOI: 10.1109/ICPR.1996.546716.

[255] Barret Zoph; Quoc V. Le. “Neural Architecture Search with Reinforcement Learning”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[256] Barret Zoph; Vijay Vasudevan; Jonathon Shlens; Quoc V Le. “Learning Transferable Architectures for Scalable Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 8697–8710.