Hierarchical Classification of Design Decisions using pre-trained Language Models

Master's Thesis of

Janek Speit

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr.-Ing. Anne Koziolek
Second reviewer:	Prof. Dr. Ralf Reussner
Advisor:	M.Sc. Jan Keim
Second advisor:	M.Sc. Tobias Hey

15. August 2022 – 15. February 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

Software architecture documentation (SAD) is an integral artefact emerging from the development process of a software project. The SAD contributes to the ongoing success of a software project by tracking important design decisions, ensuring a shared understanding of them and forestalling software erosion. In order to improve the quality of SADs and to support downstream tasks, an automatic classification of these design decisions is desirable.

In this thesis, we implement and evaluate an approach to automatically identify and classify design decisions based on a fine-granular taxonomy by combining a hierarchical classification strategy with the exploitation of transfer learning through pre-trained language models. The main contribution of this study is to investigate the benefit of the hierarchical classification strategy for the classification of design decisions over a non-hierarchical approach. Additionally, we examine and compare the effectiveness of the pre-trained language models RoBERTa, XLNet, BERTOverflow, and GPT-3 for this task.

In our evaluation, the approaches using pre-trained language models generally outperformed the baseline approaches. However, we could not find a clear advantage of the hierarchical approaches over the non-hierarchical approaches in combination with the language models. Ultimately, the results of this thesis are limited by the size and imbalance of our data set and therefore require further research on a larger data set.

Zusammenfassung

Die Software-Architektur Dokumentation (SAD) ist ein integrales Artefakt eines Software Projektes. Die SAD trägt zum fortwährenden Erfolg eines Software Projektes bei, indem sie ein gemeinsames Verständnis der Software Architektur gewährleistet, wichtige Entwurfsentscheidungen dokumentiert und einer Erosion der Software vorbeugt. Um die Qualität von SADs zu verbessern und nachgelagerte Aufgaben zu unterstützen, ist eine automatische Klassifizierung dieser Entwurfsentscheidungen erstrebenswert.

In dieser Arbeit implementieren und evaluieren wir einen Ansatz zur automatischen Identifikation und Klassifizierung von Entwurfsentscheidungen auf der Grundlage einer feingranularen Taxonomie, bei der wir eine hierarchische Klassifikationsstrategie mit dem Einsatz von Transfer-Lernen durch vortrainierter Sprachmodelle kombinieren. Der Beitrag dieser Arbeit besteht darin, den Vorteil einer hierarchischen Klassifikationsstrategie für die automatische Klassifikation von Entwurfsentscheidungen gegenüber einem nicht-hierarchischen Ansatz zu untersuchen. Außerdem untersuchen und vergleichen wir die Effektivität der vortrainierten Sprachmodelle RoBERTa, XLNet, BERTOverflow und GPT-3 für diese Aufgabe.

In unserer Evaluation schnitten die Ansätze mit vortrainierten Sprachmodellen im Allgemeinen besser ab als die Baseline-Ansätze. Wir konnten jedoch keinen klaren Vorteil der hierarchischen Ansätze gegenüber den nicht-hierarchischen Ansätzen in Kombination mit den Sprachmodellen feststelle. Letztlich sind die Ergebnisse dieser Arbeit durch die Größe und das Ungleichgewicht unseres Datensatzes limitiert und erfordern daher weitere Forschung mit einem größeren Datensatz.

Contents

List of Figures

List of Tables

1 Introduction

Software development is a complex process, with many participants involved and many resulting information and artefacts. For the success of a software project and the resulting software, it is important that a shared understanding and clear communication is enabled. In addition, information must be preserved to ensure adequate software quality and maintenance throughout the life-cycle of a software system. To meet these requirements, good documentation is fundamental. [47]

A key role in the software development process is played by design decisions. They can be understood as a general term for decisions that affect different views or aspects of a software system. Jansen et al. [28] conclude that the software architecture can be considered as the set of all design decisions. Hence, they are of particular importance for the resulting software. Since design decisions constitute fundamental aspects of the software, their documentation is of major influence on the software development life cycle. This documentation typically comes in the form of software architecture documentation (SAD) written in unstructured natural language.

Since design decisions are a very broad term, it makes sense to categorise them further. For instance, Kruchten [37] has proposed a taxonomy for design decisions in which they distinguish at the highest level between existence decisions, property decisions, and executive decisions.

Classification of design decisions is useful for several reasons. It promotes communication and understanding of the documentation, helping those involved in the software development process to better understand design decisions. Beyond that, the classification of design decisions also supports a variety of downstream tasks, especially with respect to automated processing.

One of these tasks is automatic consistency checking between software architecture and informal documents, such as SADs. Keim et al. [32] substantiate the need for an automated consistency checking and motivate their future goal for automatically checking the consistency between models and SADs based on natural language understanding. A SAD with classified design decisions would be one step closer to that goal since the classification helps significantly in properly interpreting the meaning of a design decision. For example, if a design decision is assigned to an existence decision or, more precisely, to a structural decision, it can be specifically checked whether this structure is represented in the model. In practice, SADs are commonly written in natural language and in a form that tends to be unstructured. Although the benefit of classifying design decisions within a SAD is well reasoned, it does not take place during the documentation process. The reason for that is not least because accurate documentation already suffers from a lack of human resources, time, and money. Subsequent manual classification would also be very costly and require trained personnel. As a consequence, we see the demand for an automated classifier. The aim of this thesis is, therefore to develop and evaluate an approach that identifies design decisions of a SAD and classifies them according to the taxonomy proposed by Keim et al. The addressed goal is a text classification task from the field of natural language processing (NLP). For this purpose, *conventional* methods exist, in which a preprocessing of the texts with techniques of a conventional NLP pipeline is performed. In recent years, the increasing availability of computational resources has spurred the research and popularity of pre-trained language models (LM) that provide promising results in various NLP tasks. These modern LMs provide the advantage of being pre-trained on a large text corpus. This pre-knowledge can be exploited for NLP tasks in addition to the actual training data. This process is also referred to as transfer learning. We consider this property of LMs to be particularly valuable for our case since we have a very limited amount of labelled data available for training. We can therefore expect LMs to have an advantage over conventional approaches. Thus, one goal of this work is to systematically apply different state-of-the-art LMs and compare them with respect to their classification performance.

The taxonomy that is used as a basis of the classification has a hierarchical structure. On the one hand, a classification can be done flat by ignoring the hierarchy and only considering the leaf node classes. On the other hand, the hierarchy can be incorporated for classification.

This leads us to the research question of whether a hierarchical approach yields better classification results than a flat approach. The assumption is that dividing the problem into smaller sub-problems and thereby incorporating the hierarchical information into the classification task enables a better classification performance. Therefore, we want to investigate whether this assumption is confirmed. Furthermore, we want to analyse the performance of each parent class to identify over- and under-performing classes.

In summary, the main contribution of the master's thesis is to investigate the hierarchical classification of design decisions using pre-trained LMs and to answer the following research questions:

RQ1: Are there parent classes or hierarchy levels in the taxonomy at which classification performance is proportionally high or low?

RQ2": What is the effect of a hierarchical classification approach versus a flat approach on the overall classification result, with conventional methods and with the transformer approaches?

RQ3: How and to what extend do pre-trained LMs improve classification performance over conventional approaches?

RQ4: Which of the applied LMs leads to the best classification results?

1.1 Structure of the Thesis

The thesis is structured as follows:

In Chapter 2, we introduce the foundations of the work. We first describe the taxonomy on which our approach is based. Next, we introduce the natural language processing techniques and algorithms as well as the transformer-based language models. At the end of this chapter, we define the performance metrics used for the evaluation. In Chapter 3, we present an overview of related work that investigates the automatic classification of design decisions, the use of pre-trained language models in software engineering-specific tasks and hierarchical text classification. This is followed by chapter 4, in which we introduce our approach in more detail. In this chapter, we describe the hierarchical classification strategy. Further, we explain the implementation of the language model-based approaches and the baseline approaches. In Chapter 5, we go into the evaluation by first presenting our evaluation methods, the data set and a GQM plan. Then we present and examine our evaluation results in detail. Finally, Chapter 6 concludes the thesis. In the conclusion chapter, we first summarise the evaluation results and revisit the research questions, followed by a discussion of threads to validity and future work.

2 Foundations

This chapter introduces the essential techniques and methods of machine learning, especially natural language processing (NLP), that we apply in this work. In the first section, we introduce the taxonomy based on which we aim to classify the design decisions. Next, we explain methods that we refer to as *classical* methods, which we have applied to our baseline classifiers. After that, we explain the general functionality of transformer models and the transformer language models we utilise in our work. Finally, we introduce performance metrics commonly used for classifier evaluation and that we also use for our evaluation.

2.1 Taxonomy

Software architecture documentation (SAD) captures a variety of different design decisions. In order to classify these design decisions adequately, we require a suitable classification scheme or taxonomy. A study that has been deeply devoted to the development of a taxonomy for design decisions is *"A Taxonomy for Design Decisions in Software Architecture Documentation"* and was published as a preprint by Keim et al. [33]. As their main contribution, Keim et al. developed and evaluated a comprehensive and fine-grained taxonomy, as shown in Figure 2.1 and 2.2.

As the authors state in their preprint, existing classification schemes did not meet their requirements because they were usually too coarse-grained and thus impractical for specific downstream tasks such as automatic consistency checking between SAD and other software artefacts. Following this argumentation, we use the taxonomy developed by Keim et al. in this thesis.

The taxonomy categorises design decisions into several sub-classes, creating a hierarchical tree with a maximum depth of 5 and 24 distinct leaf node categories. Our objective is to classify design decisions into exactly one of these 24 classes. In addition, it is part of our classification goal to identify design decisions, i.e. to classify *if* a text line contains a design decision or not. Consequently, we see *no design decision* as an additional class leading to 25 final leaf note classes.

2.2 Classical Text Classification

We develop classifiers that rely on classical NLP or text classification techniques to provide a baseline for comparing and evaluating transformer-based text classification approaches. With classical techniques, we refer to those that have a long history of being established and researched, and do not involve the use of pre-trained language models or neural

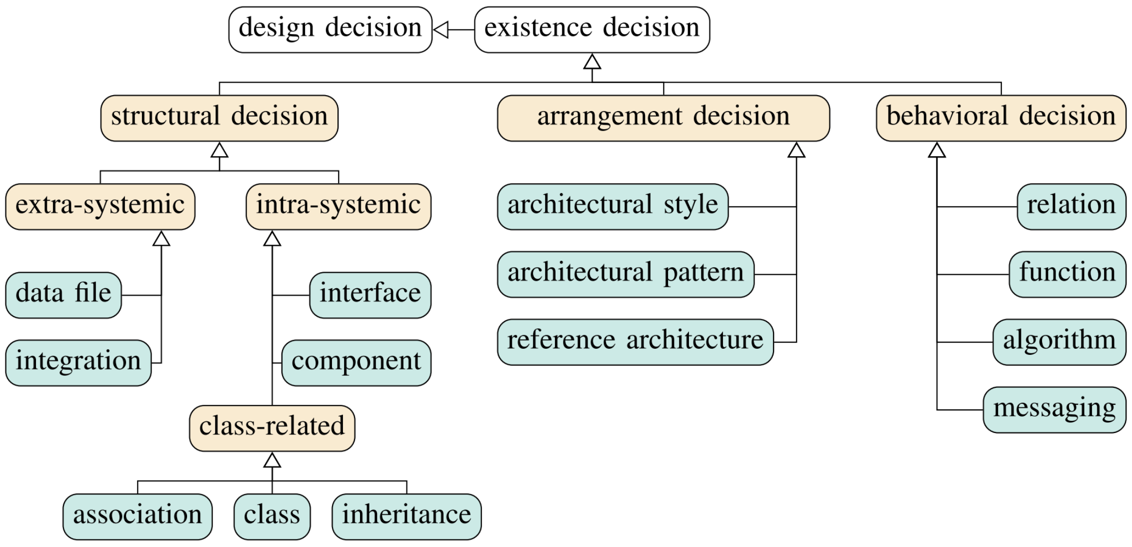

Figure 2.1: Part one of the taxonomy: sub-classes of Existence Decision (Source: Keim et al. [33])

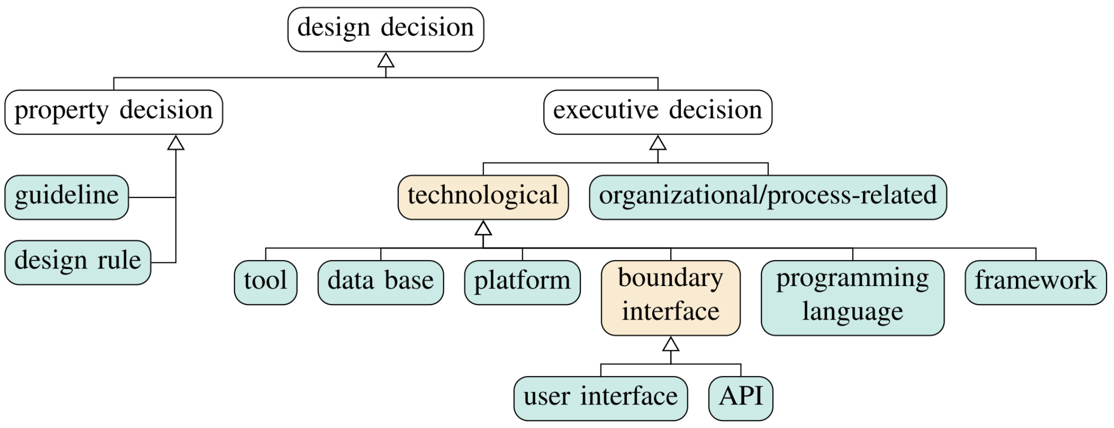

Figure 2.2: Part two of the taxonomy: sub-classes of Existence Decision and Property Decision (Source: Keim et al. [33])

networks. The classical process of automatic text classification can be broadly broken down into four pipeline steps: 1. preprocessing, 2. tokenization, 3. feature engineering, and 4. training and evaluation of the classifier. We will describe these steps and their techniques in this section.

2.2.1 Preprocessing

In NLP, preprocessing refers to a set of techniques used to clean, transform, and prepare text data for further analysis or modelling. The goal of preprocessing is to make the text data more amenable to analysis and to remove any noise or irrelevant information that may interfere with subsequent NLP tasks, such as text classification.

Text Cleaning Some preprocessing steps include cleaning the text. This can involve removing punctuation, other undesirable characters or stop words. Stop words are a set of commonly used words, such as *"the"*, *"is"*, *"and"*, *"in"*, that do not add significant meaning to a text.

Stemming & Lemmatization Stemming and lemmatization reduce inflected or derived words to their base form. For example *"developing"* gets reduced to *"develop"*. Stemming usually applies simple rules and heuristics to remove the inflectional ending of a word. Lemmatization, on the other hand, is more precise, typically using a vocabulary with morphological analysis and incorporating the syntactic structure of the sentence.

2.2.2 Tokenization

Tokenization is an integral step in the NLP Pipeline. It describes the process of breaking down a piece of text into smaller units called tokens. In classical NLP approaches, tokens are usually represented by single words or characters. Depending on the type of token, tokenization is then performed using separation patterns or simple rules. In novel applications, such as transformer-based language models, subword tokenization has been established. Subword tokenization learns a vocabulary of frequently occurring substrings based on a large text corpus. As a result, frequently occurring words are represented by a single token, while rarer or unknown words are composed of several tokens [2]. New texts are then tokenized by matching the substrings of the text with tokens in the vocabulary. The advantage of this method is that subword tokens have greater expressiveness than single-character tokens while minimizing the problem of dealing with unknown words. Popular subwort tokenization algorithms are for instance WordPieces [67], Unigram [38] or Byte-Pair Encoding [53].

2.2.3 Feature Engineering

The goal of feature engineering is to map the preprocessed and tokenized text to numerical vectors used to train the classifier in the subsequent step. The entries of the vectors represent numerical information about the text, also referred to as features. Accordingly, the vectors are also known as feature vectors and their vector space as feature space. Common feature engineering methods techniques are describe below.

Bog-of-Word The bag-of-words approach is a simple method to embed text into a feature space. A text is considered a set or multiset of words. Thus, word occurrences are taken into account, but not their order. In bag-of-words, each word in the training corpus is assigned a feature representing its occurrence. The value of the feature can then be binary, representing a boolean value, or measuring its absolute or relative frequency in the document.

N-Grams n-grams are another method to extract features from texts. This method considers the occurrences of tuples of n words (word n-grams) or n characters (character n-grams). Analogously to the bag-of-words approach, each n-gram is assigned a feature that quantifies its occurrences in binary terms or as relative or absolute frequencies. In contrast to the bag-of-words approach, n-grams partially preserve the order of words and thus acquire more text structure.

Tf-IDF The tf-idf stands for term frequency - inverse document frequency and is a statistical measure that rates the relevance of a term to a document. A term can be, for instance, a single word or an n-gram. The tf-idf of a term t with respect to a document d from a text corpus D is calculated as follows:

$$\text{tf-idf}(t, d) = \text{tf}(t, d) * \text{idf}(t, D)$$

Where $\text{tf}(t, d)$ is the relative frequency of t in document d and $\text{idf}(t, D)$ is the inverse document frequency of t with respect to a text corpus D, defined as:

$$\text{idf}(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

It follows that the idf becomes large if t occurs in only a few documents. The intention behind the tf-idf is that not only the frequency of a term in a document measures its relevance, but also its exclusivity. A term that also occurs in many other documents is not particularly characteristic for the regarded document.

2.2.3.1 Feature Selection

Many feature engineering approaches struggle with extracting too many irrelevant features that result in an overly high-dimensional feature space. This phenomenon is called the curse of dimensionality [1], [54]. The curse of dimensionality causes the data points to become widely dispersed and sparse, making it difficult for the classifier to identify patterns or relationships between features that can be used to distinguish between different classes. This can lead to poor performance, such as overfitting or underfitting, and make it difficult for the classifier to generalize to new data.

This problem can often be insufficiently curbed by preprocessing steps such as stemming, stop-word removal or lower-casing. Applying statistical feature selection techniques can address the curse of dimensionality [26] These methods use statistical tests to evaluate the relationship between each feature and the target variable. Features with a high correlation or statistical significance are selected, while those with a low correlation or statistical significance are discarded. The following feature selection methods are applied in the context of this work.

Mutual Information The first method is based on mutual information [57]. Mutual information, in simple terms, scores how much information can be gained about a target variable by observing another variable. Thus, for feature selection, the features can be ranked according to the amount of information they carry about the target class.

ANOVA F-Value Another metric for feature selection utilises the ANOVA f-value [17]. The f-value tests how well a feature discriminates between target classes by comparing their class distributions with respect to a feature. A feature where the class distributions are well separated and have little or no overlap discriminates well between classes and thus results in a high f-value.

Chi^2 The third feature selection method is based on the Chi^2 test [69]. This test measures the dependence of two variables. This allows us to select features on which our target class depends the most.

All of these methods have in common that they rank features using the aforementioned information-theoretic or statistical metrics. By means of these metrics, the best n features can be selected, where n is a hyperparameter to be chosen.

2.2.4 Classification Algorithms

For this thesis, we involve four classification algorithms, based on Naive Bayes, random forest, logistic regression and support vector machine. In this subsection, we will briefly introduce the four basic concepts of these classification algorithms.

2.2.4.1 Naive Bayes Classifier

Naive Bayes classifiers [7] are a family of probabilistic algorithms that is based on Bayes' theorem. Together with the *naive* assumption that all features are conditionally independent of each other with respect to a class, the Bayes' theorem can be used to determine the probability of a class given a set of features

$$P(c_k|x_1, ..., x_n) = \frac{P(x_1, ..., x_n|c_k)P(c_k)}{P(x_1, ..., x_n)} \propto P(c_k) \prod_{i=1}^{n} P(x_i|c_k)$$

Where c_k is a class and $x_i \in X$ is a feature. From that, we can derive a simple classifier that predicts a class with the highest probability

$$c_k = \underset{k}{\operatorname{argmax}} P(c_k) \prod_{i=1}^{n} P(x_i|c_k)$$

2.2.4.2 Random Forest

A random forest classifier [70] is an ensemble classifier method that combines the predictions of multiple decision trees. The basic idea behind a random forest classifier is to train multiple decision trees on different subsets of the training data and then combine their predictions by taking a majority vote from all the decision trees.

A decision tree is built using a process called recursive partitioning. The process starts with the entire dataset at the root node and then repeatedly splits the data into subsets based on decision rules and the values of input features. The subsets are split until each data in the set belongs to the same class. The decision tree can then classify a new sample by traversing the tree node by node according to the decision rules until it is assigned to a leaf node corresponding to a class.

The random forest creates multiple decision trees by using a technique called bootstrap aggregating. Bootstrap aggregating randomly selects a subset of the data, with replacement, to train each decision tree. In addition, the random forest also uses a technique called feature randomness, which randomly selects a subset of features at each decision tree split. This helps to reduce the correlation between the trees.

2.2.4.3 Logistic Regression

Logistic regression [31] is a statistical regression analysis method that is commonly used for classification tasks. This method estimates the probability of a binary outcome based on a given set of independent variables. Logistic Regression models the probability of a binary outcome through the use of a sigmoid function, described by the following equation:

$$P(Y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n)}}$$

where $P(Y = 1)$ is the probability of the outcome 1, $x_1, ..x_n$ are the independent variables and $\beta_0, ..., \beta 1$ are regression coefficients. The logisic regression can be transformed into a linear model, that models the log-odds of the outcome as a linear combination of the independent variables.

$$\ln\left(\frac{P(Y = 1)}{1 - P(Y = 1)}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

As a result, the regression parameters $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$ can be determined analogously to a linear regression using maximum likelihood estimation or gradient descent.

The primal logistic regression models only binary classes, but can be adapted in different ways for multi-class settings. This is known as multinomial logistic regression. A simple solution for multinomial logistic regression is to set up and combine one logistic regression model per class in a one-vs-rest fashion.

2.2.4.4 Support Vector Machine

Support vector machines (SVM) [45] are a family of machine learning algorithms for regression and classification tasks. The original SVM was developed for binary classification

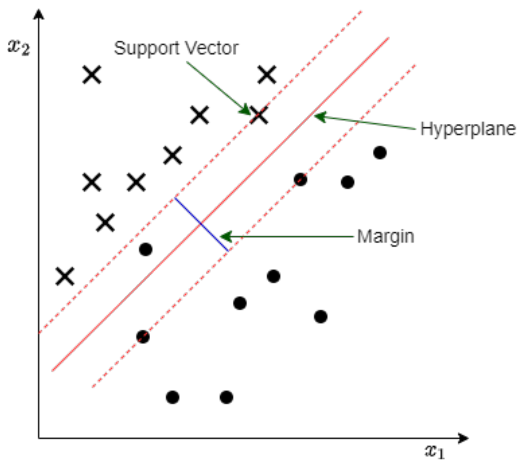

Figure 2.3: Illustration of the basic support vector machine

but has since been adapted for various applications, including multi-class classification and regression tasks. The concept of SVM is to find a hyperplane that best separates data points in the feature space according to their class. The objective is to place the hyperplane so that the margin between the two classes is maximized. The margin describes the sum of the distances between the hyperplane and the nearest data point of each class, called support vectors. Since the data points usually cannot be perfectly separated due to outliers, a soft margin is used, a modification to the margin to ignore a certain number of outlying data points. Figure 2.3 illustrates the basic SVM concept.

Using a hyperplane to separate the classes, the ordinary SVM implements a linear classifier and is thus strongly limited in its capabilities. However, the SVM can also perform non-linear classifications while preserving the maximum margin hyperplane objective by applying a technique known as the kernel trick. By applying a non-linear kernel function, the data points are transformed into a higher dimensional feature space, in which the data points then become separable by a hyperplane.

2.3 Transformer-based Language Models

Novel NLP approaches increasingly build on language models [43]. A language model is a statistical model trained to represent the probability distribution over word sequences of natural language. Thus, for any word sequence of length n, a language model assigns a probability $P(w_0 ... w_n)$ [30]. The idea behind a language model is to have a model that can generally *"understand"* language. The language understanding can then be used as a basis for any NLP tasks. Such a concept is also known as transfer learning. The advantage of transfer learning over classical methods is that tasks do not require completely new models to be trained from scratch. This potentially improves the quality of the model and reduces the need for training data.

Modern language models are usually implemented as neural networks trained on large text corpora to learn the probability distribution over token sequences. The structural characteristics of natural language and the requirement to train on extensive amounts of

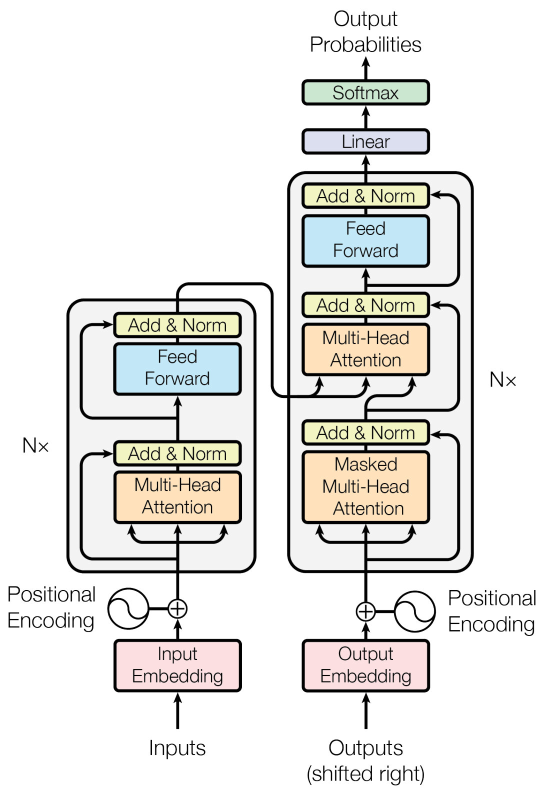

Figure 2.4: The transformer architecture (Source: Vasvani et al. [61])

data efficiently demands particular architectures for neural networks. One architecture that has emerged as the most suitable and thus became widely established is the transformer architecture.

2.3.1 Transformer Architecture

The transformer architecture was first presented by Vaswani et al. in the paper "Attention is all you need" in 2017 [61], initially as a sequence-to-sequence transformation model for language translation. In the following, we provide an overview of the main building blocks of the transformer architecture and its functionality.

The original transformer architecture consists of an encoder and a decoder part, each with a stack of encoder and decoder layers, respectively. Figure 2.4 illustrates the architecture with the encoder part on the left grey box and the decoder on the right grey box. The figure shows a single encoder and decoder layer. In the transformer model, these layers are repeated Nx times, $Nx = 6$ in the original paper.

2.3.1.1 Encoder

The role of the encoder is to map the tokens of the input sequence to an abstract contextualised vector representation, which represents the token with all its semantic and syntactic relations to other tokens of the input sequence. For this, the input sequence is processed successively by the six encoder layers. The output of the final encoder layer, i.e. the contextualised vector representations, is needed afterwards by the decoders to generate the output sequence.

Input Embedding & Positional Encoding The first step in the encoder process is to map the input sequence tokens into vector representations referred to as input embeddings. These input embeddings project the tokens into a 512-dimensional vector space. The input embeddings are determined from an embedding matrix, which learns the representations of every token throughout the pre-training.

In the next step, the input embeddings are then extended with the so-called positional encoding. Positional encoding augments the input embeddings with positional information by adding vectors to the input embeddings that encode a token's position in the input sequence. Positional encoding is a crucial feature of the transformer architecture. For the understanding of natural language, it is indispensable to consider the order of words. In models such as recurrent neural networks, the order is preserved by processing a sentence or sequence sequentially word by word. The detriment of this, however, is that models like this are hard to parallelise, which leads to a significant drawback for training with large amounts of data. Positional encoding alleviates this problem by carrying the position information of each token explicitly as part of the embeddings rather than implicitly via the structure of the input sequence. This allows long word sequences to be processed by the model at once. Consequently, transform-based language models can be trained much more efficiently, leading to the development of language models trained on extensive text corpora

Multi-Head Attention After generating the input embeddings and enriching them with positional encoding, the vectors are passed to a multi-head attention layer. The multi-head attention layer implements the self-attention mechanism, which is the second key component of the transformer architecture. Self-attention enables the model, while processing a token, to attend to all other tokens of the input sequence with different weights, capturing complex dependencies between different elements in the input without the need for explicit recurrence or convolution. The example in Figure 2.5 shows how a transformer model considers different tokens with different weights when processing the token "it". Through the self-attention mechanism, the model is thus able to correctly resolve the different co-references of "it" in the two example sentences. In the transformer architecture, self-attention is extended to be multi-headed. As such, the attention mechanism is applied by eight so-called attention heads in parallel to different representation subspaces learned from the model. Thus, different aspects of relations between words can be considered simultaneously, and relations can be modelled more precisely.

All outputs of the eight attention heads are now concatenated and linearly transformed into a lower dimension to be passed on to a feed-forward layer with a ReLU activation function, which calculates the final output of the encoder layer. In this manner, the input embeddings are fed through the six encoder layers. Eventually, the last encoder outputs contextualised vectors that represent syntactic and semantic relations of the tokens to their surrounding tokens, i.e., their context.

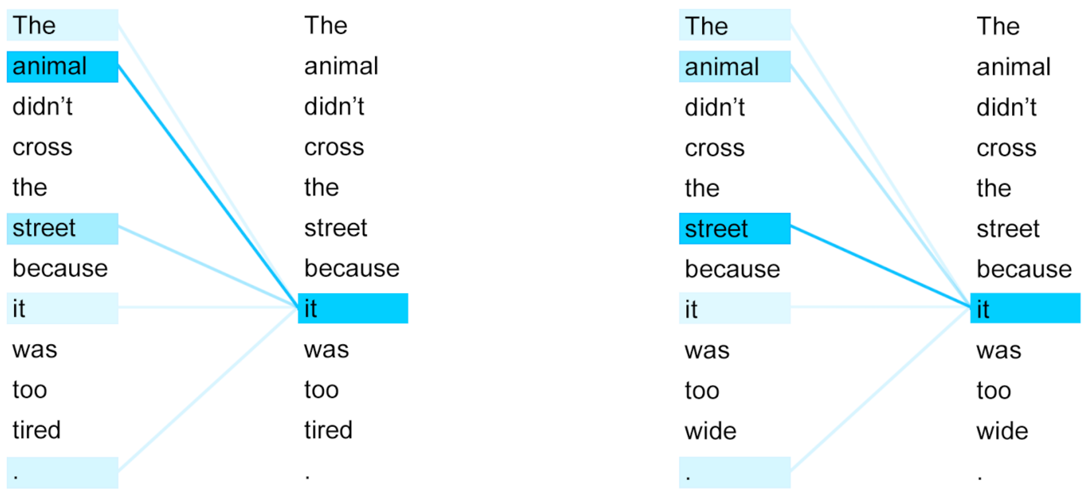

Figure 2.5: Attention while processing the word "it" (Source: Jakob Uszkoreit [60])

2.3.1.2 Decoder

The purpose of the decoder is to generate the target sequence from the contextualised vectors calculated by the encoder. Unlike the encoder, the decoder proceeds sequentially or autoregressively, generating the next token of the target output sequence one step at a time. Once the decoder generates an output token, it is fed back as additional context to the first decoder layer and considered for the generation of the next token.

Masked Multi-Head Attention In the decoder, the target sequence is transformed into an input embedding with positional encoding, similar to the process in the encoder. The target sequence is then given to a multi-headed self-attention layer, which differs in one aspect from the layer in the encoder. For the model to autoregressively learn to generate the target sequence token by token, the context that the attention mechanism can take into account must be restricted to the context on the left side of the token to predict. Otherwise, the model could *"cheat"* by already seeing the tokens yet to be predicted. The restriction is done by masking the attention and setting the weights of the future tokens to 0. Consequently, the model does not attend to future tokens in the sequence. Vaswani et al. refer to this modified version as masked multi-head attention.

Cross-Attention The masked self-attention is followed by another attention layer known as cross-attention. The cross-attention layer has a link to the output encoder stack's final layer. This cross-attention connection provides the decoder with the representations of the previously generated tokens as well as all tokens of the source sequence in a contextualized vector representation calculated by the encoder.

The cross-attention works analogously to self-attention but differs in its input. Cross-attention takes two different input sequences as input and hence, models the relations between the tokens of the two sequences. In this case, it is the source sequence, output by the encoders, and the target sequence.

The output of the cross-attention is processed by a feed-forward layer and then passed on to the next decoder. After the output has passed through all the decoder layers, it is then processed by a final feed-forward layer, followed by a Softmax function, which outputs a probability for every token in the vocabulary, determining the next token in the target sequence.

So far, we have described the original Transformer architecture, which was developed with the intent of language translation.

On this basis, the transformer architecture has been adapted and is used in a variety of application fields, including audio processing [62], and computer vision [34]. Particularly prominent are transformer models in their role as language models, helping to achieve state-of-the-art results for a wide range of natural language processing tasks [48], [43]. This also includes the software engineering domain, as we elaborate on in section 3. We distinguish two flavours of transformer-based language models, the BERT-based and the GPT models. These two transformer language model types and the particular language models we use for our experiments are introduced in the following.

2.3.2 Bidirectional Encoder Representations from Transformers

The first category of transformer language models have their origin in a language model called BERT (Bidirectional Encoder Representations from Transformers).[16]. Since the original publication of BERT by Devlin et al., several evolutions of BERT have been developed, such as RoBERTa [42] or ALBERT[40]. These BERT-like models are improvements or further developments of the original model but follow the same basic principles of masked language modelling and the encoder-based architecture. In the following, we will first introduce the original BERT model and then describe the BERT-based models used in this work.

BERT is a language model that can be conceived as a framework that provides language knowledge and can be fine-tuned for arbitrary downstream language processing tasks. In contrast to the complete transformer architecture, the BERT architecture relies solely on a stack of encoder layers. Hence, the model does not generate a token sequence but outputs embeddings, i.e., numeric vectors representing each token in its current context.

These embedding vectors can be used for downstream tasks, for instance, by adding a task-specific classification layer on top of the BERT model and training the whole model end-to-end on specific labelled data.

BERT models are pre-trained on large unlabelled text corpora using the masked language modelling objective. Masked language modelling describes a training process in which a certain percentage of the input tokens is randomly replaced by a generic [mask] token. The model is trained to predict the original values of the masked tokens, given the context on both sides. This is done by feeding the input sequence into the BERT model and using the output of the final layer as the prediction for each masked token. The model is then trained to minimise the cross-entropy loss between the predicted and the actual token values. Masked language modelling enables the model to learn the contexts before and after a token. This is why BERT is also described as being bidirectional.

On top of masked language modelling, BERT is trained on another training objective: Next sentence prediction. In the next sentence prediction, the model receives two sentences separated by a [SEP] token as input. BERT is then trained to predict whether these sentence pairs are either sentence that immediately follows each other in the training corpus or two randomly sampled independent sentences. To aid the next sentence prediction, BERT additionally carries a so-called [CLS] token at the beginning of each sentence. The output of the [CLS] token is inferred by all the other words of the sentence. Hence, it covers the context of the entire sentence. Devlin et al. implemented next-sentence prediction to help the model capture relationships between sentences, which they deemed relevant for tasks such as question-answering or natural language inference.

In this thesis, we applied and compared three different BERT-based language models, that we introduce in the following.

2.3.2.1 RoBERTa

First, we introduce the BERT-like model RoBERTa [42]. RoBERTa stands for "*Robustly optimised BERT approach*" and is a pre-trained language model that adopts the BERT architecture unmodified.

Liu et al.'s optimisations over the original BERT mainly relate to the pre-training process, the hyperparameters and the datasets, with the training principle and masked language modelling objective remaining unchanged. The researchers experimented with various adjustments to the pre-training and, based on their findings, trained RoBERTa with essentially four optimisations: 1. Omitting next-sentence prediction; RoBERTa was trained without next-sentence prediction but instead on full sentences sampled contiguously from one or more documents, rather than segment pairs as in the original BERT. This also implies that RoBERTa was trained on longer consecutive token sequences with up to 512 tokens per sequence. 2. Dynamic masking patterns; For the masked language modelling, ten different random masking patterns are used per training sample instead of the same static pattern per sample. This had a positive effect, especially in combination with the more extended training, in which one sample is seen more often. 3. Significantly more training data; the training data set of BERT was extended by 155 of uncompressed text data. Thus RoBERTa was pre-trained to a total of 161 of text data. 4. Longer training with larger batch sizes; to train RoBERTa, Liu et al. used a batch size of 8k and 500k training steps, compared to the batch size of 256 and 1M training steps for BERT, leading to prolonged training.

The resulting RoBERTa model has 24 layers, 16 attention heads and 355m trainable parameters. The model was evaluated on established NLP benchmarks with challenges including question answering and sentence classification, showing that RoBERTa clearly outperforms BERT and performs equally well or slightly better in all compared to XLNet.

2.3.2.2 XLNet

Another evolution of BERT is XLNet.[68]. With XLNet, Yang et al. developed a transformer-based language model based on a stacked encoder architecture but replaced BERT's masked language modelling with a new pre-training objective named permutation language modelling.

In BERT's masked language modelling, the authors see two main problems. One problem is that for pre-training the corruption of the input sequence is done by replacing tokens with the [Mask] token. However, this [Mask] token does not occur in real-world data, leaving an ambiguous discrepancy between pre-training and fine-tuning of the model. Further, the authors find the problem that BERT presumes conditional independence between the masked tokens of an input sequence. Suppose the sentence *"New York is a city"* is masked to *"[Mask][Mask] is a city"*. BERT may then predict *"New"* and *"Francisco"* for the masked words as BERT reduces to the objective

$$\mathcal{J}_{\text{BERT}} = \log p(\text{New} \mid \text{is a city}) + \log p(\text{Francisco} \mid \text{is a city})$$

XLNet addresses these two problems by introducing generalized autoregression as a training procedure that relies on the permutation language modelling objective. This training method is autoregressive. That is, one token is predicted based on all previous tokens, i.e. the previous context. To also consider the context behind a token, XLNet evaluates all possible factorization orders of the token in an autoregressive fashion, in contrast to BERT's fixed forward factorization order.

Figure 2.6 shows four permutations on a four-token input sequence in which the token x_3 is to be predicted. Due to the different factorization orders, all tokens occur at least once in the left context of the processed token. Note that only the factorization order is permuted. The actual token order is preserved through positional encoding.

By working autoregressively, permutation language modelling resolves the problems of masked language modelling by taking into account dependencies between the tokens to be predicted, reducing the training of XLNet in the example to the following objective:

$$\mathcal{J}_{\text{XLNet}} = \log p(\text{New} \mid \text{is a city}) + \log p(\text{York} \mid \text{New}, \text{is a city})$$

Further, the training-fine-tuning discrepancy is tackled by no longer requiring a specific mask token.

Besides improving the pre-training procedure, XLNet extends the original transformer architecture based on the publication of Transformer-XL[15] by adding a recurrence mechanism to the transformer architecture to handle longer dependencies and a larger context. In the original transformer architecture, the context is limited to the length of a segment, a fixed number of tokens that are processed at once. Thus, dependencies that exceed this segment size cannot be modelled, which may result in torn dependencies due to the segmentation.

XLNet incorporates a recurrence mechanism for processing the segments to overcome this limitation. With the segment-level recurrence mechanism, each layer obtains the representations computed for the previous segment along with the output for the current segment from the predecessor layer. This increases the largest possible dependency length by N times, where N is the depth of the network, as contextual information can now flow across segment boundaries.

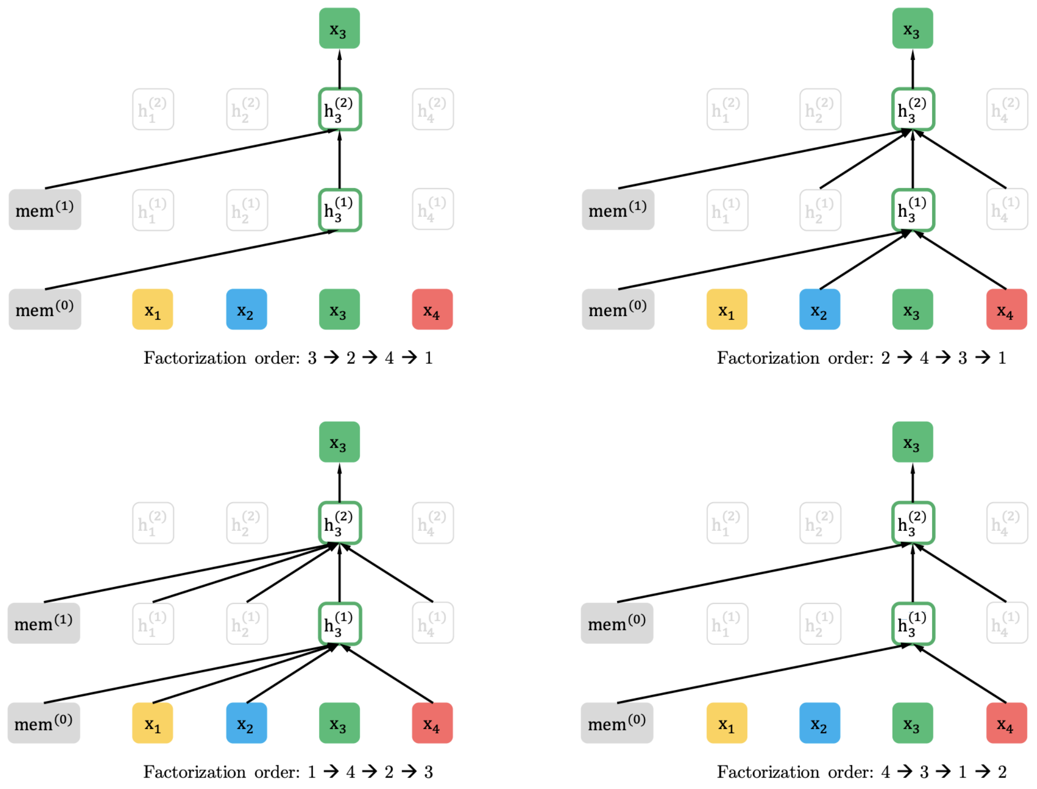
mem$^{(1)}$
mem$^{(0)}$
Factorization order: 3 → 2 → 4 → 1
Factorization order: 2 → 4 → 3 → 1
Factorization order: 1 → 4 → 2 → 3
Factorization order: 4 → 3 → 1 → 2

Figure 2.6: Permutation language modelling objective for predicting x_3 given the same input sequence x but with different factorization orders. (Source: Yang et al. [68])

2.3.2.3 BERTOverflow

The third BERT-based model we experiment with is BERTOverflow. BERTOverflow was designed as part of a study by Tabassum et al. on named entity recognition in software engineering tasks, which is also discussed in the related work section. BERTOverflow was developed to handle domain-specific language better than the original BERT, eventually improving the performance of named entity recognition for software engineering. To this end, Tabassum et al. trained a new BERT$_{BASE}$ model, which comprises 12 layers, 12 attention heads, and 110M trainable parameters, on a domain-specific new text corpus. This corpus consists of 152M sentences collected from the programmer community platform StackOverflow. The study demonstrates that BERTOverflow indeed performs better in the evaluated software engineering specific named entity recognition, despite the significantly smaller text corpus on which it was pre-trained.

Since SADs contain software engineering-specific terms or language, we apply BERTOverflow for our experiments.

2.3.3 Generative Pre-trained Transformer Models

In tandem with the BERT-based language models described so far, another flavour of transformer-based language models has emerged, namely the generative pre-trained transformer (GPT) models. As indicated by their name, the primary function of these models is to generate text.

Unlike BERT models, GPT models are based on the decoders of the transformer architecture owing to their generative nature. The decoders' layout is in line with the original transformer architecture, except for the absence of the cross-attention layer as there is no encoder input (as shown in Figure 2.7).

A GPT model is designed to predict the most probable next token based on a token sequence. For this purpose, the model is trained in an autoregressive fashion, generating one token at a time and feeding the newly generated sequence back into the initial decoder layer as context for the generation of the next token. Hence, GPT models are trained to maximise the following likelihood [49]:

$$\mathcal{J}_{\text{GPT}}(U) = \sum_{i} \log P(u_i | u_{i-k}, ..., i-1)$$

where $U = \{u_1, ..., u_n\}$ is text corpus and k is the size of the context window.

To avoid the model from seeing the future tokens to be predicted, the model is restricted by the masked self-attention to attend to only the prior context of a token. Hence, GPT models are not bidirectional in terms of capturing context.

2.3.4 GPT-3

GPT-3[11] is a generative language model introduced by OpenAI in 2020 as a follow-up to GPT-2. GPT-3 is based on the general decoder-based architecture with some modifications. The most significant modification is the integration of a sparse transformer. The sparse

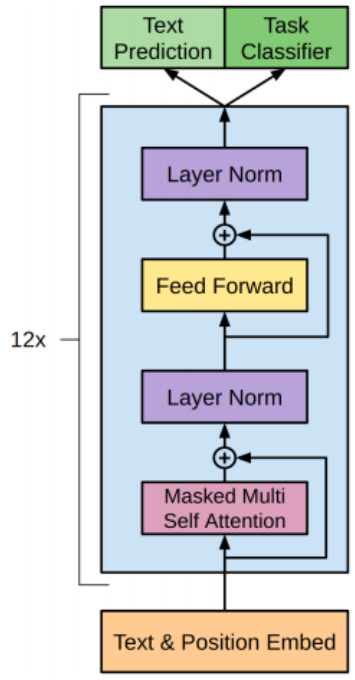

Figure 2.7: General GPT Architecture (Source: Radford et al. [49])

transformer is an extension of the transformer architecture that aims to improve computational efficiency by selectively attending to a subset of the input sequence rather than all of it. More precisely, the attention mechanism is split into two steps: a sparse attention step and a dense attention step. In the sparse attention step, the model first computes a sparse attention mask that is used to select a subset of the input sequence to attend to. This mask is computed by applying a learnable sparse attention function to the input sequence. In the dense attention step, the model performs regular multi-head attention on the selected subset of the input sequence. This reduces time and memory complexity in attention layers from $O(N^2)$ to $O(N * \sqrt{N})$ with respect to the size of the context window (in GPT-3 this equals the input sequence size).

GPT-3 stands out from other language models due to its enormous scale in terms of the network size and the training corpus. OpenAI compounded a text data set that is 570GB in compressed size and includes 500B tokens, which, assuming 0.75 words per token, is about 375B words. They split this data set into 60% training data and 40% evaluation data. The text corpus consists primarily of filtered text from the CommonCrawl data set, complemented with some smaller text corpora. OpenAI trained GPT-3 models in different sizes. The largest model includes 96 layers, 96 attention heads, and a total of 175B trainable parameters, which is about 50x more than RoBERTa or XLNet.

Zero-Shot Learning, One-Shot Learning & Few-Shot Learning The authors see the strength of GPT-3 in its flexibility to be used for different NLP tasks without fine-tuning the model with much training data. They evaluate GPT-3, therefore, with three methods, referred to as Zero-Shot learning, One-Shot learning and Few-Shot learning. In conventional fine-tuning, the model is given a larger number of task-specific examples in the form of input-output pairs, from which the model itself trains by adjusting its weights via gradient update. The drawback of this method is that a high number of labelled training data must be available. In contrast, the model can be given solely a task description, from which the model infers the expected output. For instance, for text summarisation, a prompt could be the text itself along with the term *"TL;DR"*. The model may have learned from the text-corpus that the acronym *"TL;DR"* indicates a subsequent brief summarisation of the antecedent text and hence, will generate a text that summarizes the given input. Since this example is provided without any labelled training samples, it is referred to as *"zero-shot learning"*. For instance, language translation could be prompted through the following input text:

```
Translate English to French
cheese =>
```

In this case, GPT-3 would probably continue the text sequence with the word *"fromage"*, i.e. the French translation of cheese. Since this example is provided without any labelled training samples, it is referred to as *"zero-shot learning"*. To improve the result, the input can be extended with task examples. In this case, a task description is not necessarily required. An input for sentiment analysis could look like the following, for example:

```
Comment: "I hate it when my phone battery dies."
Sentiment: Negative
###
Comment: "My day has been a mess"
Sentiment: Positive
###
Comment: "This is the link to the article"
Sentiment: Neutral
###
Comment: "This new music video was incredibile"
Sentiment:
```

Based on the given structure and the examples in the input text, the model can predict that one of the labels *"Negative"*, *"Neutral"* or *"Positive"* must follow. The GPT model will then output the most likely of these three words based on its pre-trained language knowledge. Because in this example, very few labelled training examples are provided, this method is called *"few-shot learning"* or *"one-shot learning"* in the case of exactly one training example.

OpenAI currently deploys GPT-3 as a commercial product in four different versions. The four models, Ada, Babbage, Curie, and Davinci, are described by OpenAI as having different capabilities and speeds and are priced individually. OpenAI does not publish the exact hyperparameters of the models. Based on the descriptions, we assume that Ada is the smallest model, followed by Babbage, Curie and Davinci as the largest model.

The models can be accessed via an API provided by OpenAI. Moreover, the API provides an interface for fine-tuning individual instances of the models. However, details on the fine-tuning process or implementation are not publicly available.

2.4 Evaluation Metrics

To evaluate the performance of the classifiers with regard to the classification of a single class, we resort to the performance measures precision, recall and f1-Score. These are defined as follows:

$$P_x = \frac{tp_x}{tp_x + fp_x} \qquad R_x = \frac{tp_x}{tp_x + fn_x} \qquad F1_x = \frac{2 * P_x * R_x}{P_x + R_x}$$

Where the tp_x is the number of true positives of class x, fp_x is the number of false positives, and fn_x is the number of false negatives regarding class x. For the classification performance with respect to all classes X of a classifier, we consider macro-averaged performance measures:

$$P_{macro} = \frac{\sum_x P_x}{|X|} \qquad R_{macro} = \frac{\sum_x R_x}{|X|} \qquad F1_{macro} = \frac{\sum_x F1_x}{|X|}$$

Additionally, we measure the classifier's accuracy, defined as follows:

$$A = \frac{\sum_x tp_x}{s}$$

where s is the total number of samples. Note that in the multi-class setting, the accuracy is equivalent to the micro-averaged F1 score [24].

Further, we calculate Matthews correlation coefficient (MCC) for each classifier we train. The MCC measures the correlation between predicted classes and true classes. We use a modified version of the MCC adapted to the multi-class setting, as proposed by J. Gorodkin [23]. The multi-class MCC is defined as:

$$MCC = \frac{c * s - \sum_x p_x * t_x}{\sqrt{(s^2 - \sum_x p_x^2) * (s^2 - \sum_x t_x^2)}}$$

where

- $t_x = tp_x + fn_x$ being the number of times class x occurs
- $p_x = tp_x + fp_x$ being the number of times x was predicted
- $c = \sum_x (tp_x + tn_x)$ being the total number of samples correctly predicted
- s being the total number of samples

Alongside the performance measures mentioned so far, we consider hierarchical performance measures to evaluate the final classification result. These are modifications of the common measures to a hierarchical classification scenario to not only discriminate true and false classifications but also take into account the distance between the correct and the predicted class in the class hierarchy. The hierarchical performance metrics are defined below:

$$hP = \frac{\sum_i |P_i \cap T_i|}{\sum_i |P_i|} \qquad hR = \frac{\sum_i |P_i \cap T_i|}{\sum_i |T_i|} \qquad hF1 = \frac{2 * hP * hR}{hP + hR}$$

Where P_i is the set consisting of the predicted leaf node class for test example i, and all its ancestor classes in the taxonomy, analogously, T_i is the set consisting of the true leaf node class for i and all its ancestor classes.

3 Related Work

This section aims to provide an overview of existing research related to this work in one or more dimensions. In particular, we contemplate three aspects.

At first, we consider work dealing with automated classification of (design)- decisions. Next, we explore work that uses pre-trained transformer-based LMs for text classification in the Software Engineering (SE) domain. Finally, we give an overview of research that addresses hierarchical text classification, especially those that employ the local classifier per parent node (LCPN) strategy.

3.1 Automatic Classification of Decision

Existing research on automated classification of design decisions, or decisions in general, uses different premises, information sources, and classification schemes to extract and classify decisions. In this regard, these works explore different machine-learning techniques for text classification.

A closely related work is the pre-print published by Keim et al.[33], which we have already introduced in the foundations chapter. In addition to the taxonomy presented, Keim et al. implemented approaches to identify and classify design decisions, achieving an averaged f1-score of 89.5% for identification using a random forest approach and an f1-score of 55.8% for classification of design decisions using BERT. Our thesis thus ties in with the work of Keim et al. focusing more intensely on identifying and classifying design decisions.

Bhat et al. [8] proposed automatically detecting and classifying design decisions in issues from an issue tracking system. They relied on a two-phase supervised machine learning approach to first automatically detect design decisions from issues and second, to automatically classify the identified design decisions into five different decision categories. For classifying and detecting decisions, Bhat et al. trained and compared different classifiers, such as logistic regression, SVM or Naive Bayes, on a manually labelled data set. They used a TF-IDF representation of N-grams as features. In the end, the SVM yielded the best results with an average accuracy of 91.27% for detection and an average accuracy of 82.79% for classification.

Another related work was published in 2021 by Li et al. [41] that investigates the automatic identification of decisions from textual artefacts using different machine learning techniques. They used 1300 sentences gathered from the Hibernate developer mailing list labelled either as a decision or as no decision. Li et al. experimented with different machine learning algorithms in combination with different pre-processing and feature extraction configurations by training and evaluating classifiers with 10-fold cross-validation. The evaluation indicated that an SVM with a simple bag of words provided the best results with a precision of 0.64, a recall of 0.93, and an f1-score of 0.76.

In a following publication, Fu et al. [20] followed up on this work. They proposed to classify these various types of decisions, identified from the Hibernate developer mailing list, into the five categories design decision, requirement decision, management decision, construction decision and testing decision, to aid their documentation. Fu et al. evaluated different ensemble classifiers in which multiple machine-learning methods were applied jointly in one classifier. Furthermore, they tested different pre-processing strategies, vectorisation methods and a CH^2 feature extraction algorithm. The best result with an f1-score of 0.727 was achieved by an ensemble classifier that involves linear regression, SVM and naive Bayes in combination with a simple bag of words approach for the vectorisation.

The research outlined so far relies on classical feature engineering and machine learning methods for text classification. In contrast, Joseph et al. [29] take a different tack by leveraging transfer learning through BERT. They as well motivated the need for proper documentation of design decisions. To support this task, they presented a concept and early implementation of a bot integrated with the instant communication platform Slack. The Slack bot listens to textual communication between developers to identify design decisions and classifies them into the seven WH(Y) classification template categories. Furthermore, the bot was designed interactively to improve the classification quality continuously. Once a design decision is identified, the most likely classes are suggested to the user. Hence, users can select the correct class and generate new labelled data to fine-tune the BERT model further. The early prototype implementation of their bot achieved a precision close to 1.0. However, Josephs et al. state that good results can probably be fathomed by high overfitting, as the data set so far only consists of 55 training clauses and 38 test clauses.

3.2 Pre-Trained LMs for Software Engineering Tasks

We have shown that there is already some research to identify and classify different types of decisions from textual sources. To the best of our knowledge, there is only one approach relying on pre-trained transformer LMs for this purpose. Nevertheless, there is a sundry of other text classification tasks in the software engineering domain where pre-trained Transformer LMs have been utilised, which we exemplary overview in this section.

3.2.1 Requirements Mining and Classification

Classification of requirements is a task related to the classification of design decisions. Requirements are often found in natural language texts and can be distinguished into functional and non-functional requirements or more fine-grained categories. The track record of transformer-based LMs thus also lends itself to the application for requirements classification, as indicated by recent research.

In 2020 Hey et al. [25] proposed NoRBERT, which stands for non-functional and functional requirements classification using BERT. NoRBERT was evaluated on different tasks of requirements classification such as binary classification in functional and non-functional, multi-class classification of non-functional subclasses and other classifications.

For the evaluation, hey et al. applied a project-fold cross-validation and a leave-one-project-out cross-validation in addition to 10-fold cross-validation to examine the generalisation of the fine-tuned BERT models to unseen projects. Their findings confirm that NoRBERT performs equally or better in all requirements classification tasks. The advantage of transfer learning could be played out, especially in unseen projects. For example, the project-fold validation strategies led to a 15% better f1 score compared to state-of-the-art results considering the binary classification of requirements based on functional and quality aspects on a data set with 612 requirements.

Another study leveraged software engineering contracts as a data set for extracting and classifying requirements and compared BERT with other machine learning methods for this purpose [51]. Sainani et al. likewise obtained significantly better results through fine-tuned BERT compared to other machine learning methods. They investigated the automatic extraction of requirements from software engineering texts and then the multi-class classification into 14 requirement types. In both disciplines, BERT performed better than the comparison methods SVM, Random Forest, naive Bayes and BiLSTM. Especially in multi-class classification, the BERT model outperformed the second-best approach, which used a BiLSTM, by 14% f1-score.

Requirements can be extracted from various text artefacts. One useful source is app reviews. de Araújo et al. addressed this and improved existing approaches for extracting requirements from app reviews, which so far only provide poor results, by using a transfer learning approach based on BERT [4]. Requirements extraction can be approached as a token classification problem. Tokens are labelled as beginning, middle, end or outside of a requirement phrase. de Araújo et al. fine-tuned the BERT model on this task with 1000 labelled app reviews from different apps. In their evaluation, they achieved an averaged f1 score of 46%. This is several orders of magnitude higher than the comparative approaches, which are rule-based or use linguistic patterns and come to f1 scores of 5-7%.

3.2.2 Sentiment Analysis for Software Engineering

Sentiment analysis is about categorising opinions or sentiments expressed in a piece of text, usually into positive, negative or neutral. This task has been addressed extensively in the past for application in various fields. Research on this topic suggests that sentiment analysis is particularly challenging for software engineering-specific text, which is why current research is tackling this challenge by applying pre-trained language models to this problem.

The question of whether pre-trained LMs can improve the performance of sentiment analysis for software engineering is addressed by Biswas et al. [9] as they developed a tool based on the fine-tuning of BERT, and compared it to an existing tool that employs Recurrent Neural Netorks (RNNs). They first used a data set with 1500 labeled phrases from Stack Overflow for fine-tuning and evaluation. Then they extended the data set with 4000 additional phrases, also from Stack Overflow, to investigate the impact of a more extensive data set on classification performance. The experiment results showed that the BERT-based tool achieves about 20% higher f1-score than the comparison tool. Further, it was found that the larger data set improves the f1-score of the BERT tool from around 80% to 87% compared to the smaller one.

A study by Zhang et al. points [71] in the same direction as their experiments yielded significant improvement in sentiment analysis for software engineering with BERT-based models against prevailing methods. More specifically, they experimented with the BERT-based models BERT, XLNet, Albert, and RoBERTa and compared these approaches to five established methods for sentiment analysis in software engineering. They trained and evaluated six different data sets, the smallest containing 341 and the largest with 7122 phrases. Depending on the data set, they improved from 6.5% up to 35.6% with respect to the f1-score, compared to the benchmark approaches. However, no clear winner stood out among the four BERT$_BASE$ models applied.

Similar results are also presented in later publications by Batra et al. [6] and by Wu et al. [66]. Wu et al. fine-tuned BERT on various software engineering-specific data sets and found improvements in the f1-score between 3%-21% compared to existing methods without pre-trained models. Batra et al. as well applied BERT-based models and significantly improved existing tools. Their experiments also suggest that an ensemble classifier consisting of multiple BERT-based models can slightly improve classification performance, and increasing the size of the data set through data augmentation also leads to minor improvements.

So far, we have given an overview of the use of pre-trained transformer-based LMs in two software engineering-specific tasks. We considered requirements mining and classification as a reasonably related task to design decision classification. Further, examined sentiment analysis as the use of transformer-based LMs for this task has already been much studied. In addition, there are many other text classification scenarios in software engineering where the use of transformer-based LMs has been examined, e.g., the classification of issues of an issue tracking system [27] or the classification of app reviews [3]. Beyond text classification, there are also many other NLP applications for transformer-based LMs, such as named entity recognition [58], [13].

The trend of transfer learning for language processing using transformer-based language models also spreads to the software engineering domain, for instance, for text classification problems. The research shows that these novel approaches provide promising results and encourage their usage.

3.3 Hierarchical Text Classification with LCPN

In this subsection, we will contemplate research of the second conceptual building block of this work, hierarchical text classification using the local classifier per parent node (LCPN) strategy. Due to the advance of the internet and rapid increase in textual content, automatic categorisation of those textual documents has become a relevant research topic. Topics and categories of texts can often be well arranged into upper and lower categories, resulting in a class hierarchy. It seems reasonable to include this hierarchy in the categorisation process. One way of approaching this is the LCPN method, which is investigated in this thesis and explained in more detail in section 4.1. The LCPN strategy for text and document classification has already been applied and studied in several research papers.

The first publication on this topic is by Koller et al. [36]. They aimed to train a naive Bayes classifier using only a few words as features. To do this, they experimented with a feature selection method based on information theoretical entropy. Koller et al. also compared a hierarchical LCPN classifier with a flat classifier. The approaches have been evaluated on two data sets and categories: a data set with 939 documents and a two-level hierarchy with three parent classes and two child classes each, and a data set with only 103 documents and a two-level hierarchy with two parent classes and two child classes each. The researchers found that the LCPN approach is superior, especially for configurations with very few features. The hierarchical classifier yielded up to 9% higher accuracy than the flat classifier.

An improvement in classification performance using an LCPN approach was also found by Weigend et al. [64]. They considered a data set with 13272 financial documents and a two-level class hierarchy with four parent classes and a total of 37 leaf node classes. In their research, a simple feed-forward neural network was used, and different input representations of the documents were explored. Weigend et al. evaluated both flat and LCPN classifiers and showed that the LCPN approach generally performed better with, on average, 5% better precision. Furthermore, the advantage of the LCPN approach was particularly noticeable for underrepresented classes with few examples.

Similar results were obtained in the research of Dumais et al. [18]. They classified web content into a large two-layer classification scheme with 13 parent and 150 child classes. For this purpose, they trained SVMs with different configurations and a feature selection based on mutual information on about 50k documents and evaluated them on 10k documents. The comparison between the LCPN and the flat classifier indicates that the hierarchical approach outperforms the f1-score of a flat approach by 4-6%.

Especially in the case of a large number of classes, the exploitation of the class hierarchy seems worthwhile, as suggested by a study by Gauch et al. [21]. They made their investigations on a part of the Open Directory Project, a web directory with an extensive class hierarchy. The part of the class hierarchy they used includes four levels, with 10132 leaf node classes, each with at least 31 samples. Gauch et al. applied the centroid distances to select particularly representative training documents and used them to train Rocchio classifiers for the flat and the LCPN approach. The final comparison shows that the hierarchical classifier performed much more accurately than the flat one with an accuracy of 54.5% vs 76.2% in their best cases.

More recent research by Stein et al. took up the topic of hierarchical text classification and applied more novel techniques in terms of classifiers and word vectorisations. More specifically, Stein et al. [56] experimented with fastText, XGBoost, SVM, and a Convolutional Neural Network (CNN) in combination with the word embedding methods GloVe, word2vec, and fastText. They ran the experiments on the RCV1 data set, a text classification benchmark data set with over 800k newswire stories categorised into a class hierarchy with 104 nodes (including root) distributed over four hierarchy levels. The hierarchical classifier was built according to the LCPN method. In contrast to the previous methods, documents do not have to be classified in leaf node classes but can also terminate in parent classes. To realise this, they added a virtual category for each parent class, meaning that the classification ends with the corresponding parent class if a document is assigned to it. With the LCPN classifiers, Stein et al. achieved better results in every configuration concerning the classification algorithm and word embedding as compared to the flat baseline approaches. Thus, the best hierarchical classifier achieved achieved a macro-avareged f1-score of 76.6 % and the best flat classifier only 57.7 % f1-Score.

[56]

The most recent work addressing hierarchical text classification with the LCPN strategy was published in 2021 by Brinkmann et al. [10] and, to our knowledge, is the only one so far that explored this on a transformer-based pre-trained LM. Brinkmann et al. utilised RoBERTa$_B$*ASE* to classify products into hierarchically arranged product categories based on product descriptions. They set out to investigate whether and how the classification performance improves when RoBERTa is trained with additional domain-specific language by training an extended RoBERTa model on three different text-corpora. Moreover, they trained and compared two hierarchical approaches, an LCPN approach and an approach based on a Recurrent Neural Network (RNN). The LCPN classifier works by using local classifiers per parent node to calculate the probabilities of each classification path in the tree and then choosing the leaf node class of the classification path with the highest probability using the Softmax function. In the RNN classifier, the classification is done sequentially from the root to the leaf node class. To do so, the pooled output of RoBERTa is extended with a hidden state and recurrently fed through the RNN classification head for each classification step until a leaf node class is reached. As a baseline, Brinkmann et al. implement a flat classifier with a simple one-layer classification head. The different classifiers were evaluated on two data sets containing product offer descriptions. One data set with about 13k samples, 396 nodes and an average hierarchy depth of 3 and one with about 768k samples, 410 nodes and an average hierarchy depth of 2.44. Brinkmann et al. chose the hierarchical f1 score (hF1) as one of the metrics, which we also consider in our work. The experiments of Brinkmann et al. found that the extension of RoBERTa with domain-specific language hardly improves the flat approach with default RoBERTa. The Hierarchical approach improved by only 1% hF1 score. With respect to the hierarchical classifiers, the RNN classifier performed marginally better than the LCPN classifier, with 88.98% vs 88.6% hF1. The flat baseline classifier, however, performed only slightly worse than the hierarchical ones, with an hF1 score of 87.7%.

The research on related work for hierarchical LCPN text classification reveals several papers that compare very different data sets and class hierarchies with the performance of LCPN classifiers versus flat classifiers as baselines. The results consistently show an advantage for the LCPN strategy, albeit to different extents.

4 Approach

Software architecture documentation (SAD) is an integral artefact emerging from the development process of a software project. The SAD contributes to the ongoing success of a software project by tracking important design decisions, ensuring a shared understanding of them and forestalling software erosion. In order to improve the quality of SADs and to support downstream tasks, an automatic classification of the documented design decisions is desirable, as motivated in more detail before. In this thesis, we implement and evaluate multiple approaches to automatically classify design decisions. These approaches are based on two main building blocks. The first building block is the implementation of hierarchical classifiers that utilise the hierarchical nature of the classification scheme in the classification process. The second building block is the exploitation of transfer learning by fine-tuning pre-trained transformer language models (LMs) on the design decision classification task.

In this chapter, we describe our approach in more detail. We start by explaining the hierarchical classification strategy in section 4.1. In section 4.2, we describe the classification using pre-trained transformer LMs. And finally, in section 4.3, we describe the development of baseline approaches.

4.1 Hierarchical Text Classification

Classifying design decisions using the taxonomy of Keim et al. is a tricky endeavour. The reason for this is the fine granularity of the taxonomy and the resulting large class output space with 25 leaf node classes (including the *No Design Decision* class). Furthermore, there are fine distinctions in the taxonomy between closely related design decisions, which could make classification more difficult. For example, the *Arrangement Decision* class is further differentiated into *Architectural Style*, *Architectural Pattern* and *Reference Architecture*, which describe similar decisions regarding the architecture. In order to better cope with these challenges, we propose incorporating the tree-like and, thus, hierarchical structure of the taxonomy as additional information into the classification process. Therefore, as this thesis's first building block, we apply and investigate the concept of hierarchical text classification. More specifically, we develop hierarchical classifiers according to the local classifier per parent node (LCPN) strategy [55].

The idea behind a hierarchical LCPN classifier is to divide the classification process into multiple sub-steps in a divide-and-conquer manner and, thus, to perform the classification with several classifiers, referred to as local classifiers. Accordingly, a local classifier is trained for every internal node class in taxonomy, classifying its direct child classes. This is in contrast to a "*flat*" global classifier that directly distinguishes between all leaf node classes, disregarding the hierarchical nature of the taxonomy. Figure 4.1 provides an

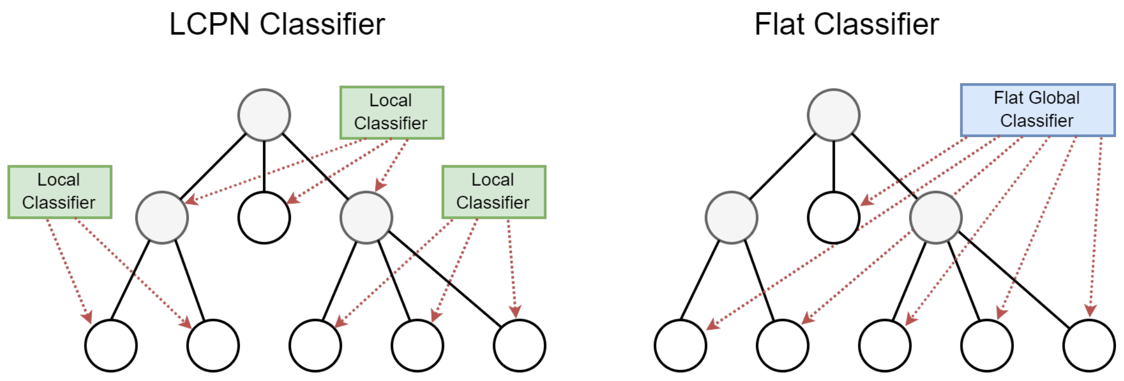

Figure 4.1: LCPN classifier vs flat classifier

example of an LCPN approach and a flat approach for a class hierarchy with six leaf node classes.

The classification with a hierarchical LCPN classifier proceeds top-down, from the root to the final leaf node class, similar to a decision tree. For the classification of a SAD text line, consider, for example, the following sentence from the SAD of TEAMMATES: *"TEAMMATES is a Web application that runs on Google App Engine (GAE)."*. This sentence contains a design decision regarding the platform. The target class of this sentence is thus *Platform*. As shown in the taxonomy in Figure 2.2, the *Platform* class has three ancestor classes. Therefore, following the LCPN strategy, the classification is done by four local classifiers. The first classifier decides whether a design decision is given (we refer to this as the *root* classifier). The local classifier for*Design Decision* distinguishes between *Property Decision*, *Executive Decision* and *Existence Decision*. The third local classifier distinguishes between the two child classes of *Executive Decision* until finally, the fourth classifier determines the final class *Platform* from all child classes of *Technological*. For hierarchically classifying design decisions according to the taxonomy, the LCPN classifier must comprise 13 local classifiers, including the root classifier.

An advantage of the hierarchical LCPN approach is that each local classifier has a significantly reduced class-output space compared to a single flat global approach. In our case, this is mostly two to three, and at most four classes. Therefore, the local classifiers are more specialised, leading us to expect improved classification of finer-grained categories, such as the classes *Architectural Style*, *Architectural Pattern*, and *Reference Architecture*, and improved performance overall. This property suggests the hierarchical LCPN approach being superior to a flat global classifier. One argument against this, however, is that for each classification, multiple steps are necessary to determine the final class. A single classification mistake from one of the local classifiers leads to a wrong final leaf node class. Therefore, we develop both the LCPN approach and a flat global approach to compare the two approaches and examine the potential added value of a hierarchical classifier.

4.2 Pre-trained Transformer-based Text Classifiers

The second building block of this work is the application of pre-trained transformer language models (LMs) for text classification. Pre-trained LMs have recently proven to be successful in a variety of natural language processing tasks. Their key advantage lies in their ability to leverage prior language knowledge through transfer learning for various NLP tasks, resulting in good performance without the need for a large data set. We see this feature of transfer learning as particularly beneficial for our specific application. Since we strive for a classification into 25 classes, the average number of training samples per class is comparatively small and reduced compared to a classification with fewer classes. Hence, we aim to tackle this predicament by exploiting transfer learning with pre-trained transformer LMs.

4.2.1 Choice of Transofmer Language Models

Given the promising nature of transformer LMs, numerous such models have been developed to date. These models vary in their size, pre-training, and other architectural and functional aspects. We choose four models from the range of transformer LMs, three BERT-based models and one GPT model, focusing on the models' capabilities rather than computational or memory efficiency. We apply the following four transformer LMs:

RoBERTa As the first BERT-based models, we apply RoBERTa-Large[42]. RoBERTa adopts the encoder architecture and masked language modelling objective from BERT, but optimises its training procedure with respect to the training hyperparameters and text corpus. As a result, RoBERTa was trained longer and on significantly more text data compared to BERT. The RoBERTa training corpus consists of a total of 161 GB of uncompressed text, including English *Wikipedia*, the *BooksCorpus*[73] with over 10k books, *CC-News*[44] with 63M English news articles, *OpenWebText*[22] with web content from various websites, and *Stories*[59], a subset of *CommonCrawl*[14] filtered to match a story-like style. The RoBERTa-Large model comprises 24 layers, a hidden size of 1024, 16 attention heads, and 355M trainable parameters.

RoBERTa achieved state-of-the-art results on popular natural language understanding benchmarks such as GLUE [63], RACE[39] and SQuAD[50]. Furthermore, RoBERTa has already been successfully used for software engineering-specific text classification tasks, as we explored in our related work section 3. Therefore, we consider RoBERTa a suitable candidate for the classification of design decisions.

XLNet Another BERT-based model we use and evaluate for the classification of design decisions is XLNet-Large, the large version of XLNet [68]. XLNet is built on an encoder-only transformer architecture and implements a new permutation language modelling objective instead of BERT's masked language modelling approach. In addition, XLNet's transformer architecture incorporates a segment-level recurrence mechanism that enables the modelling of longer contexts beyond the segment boundary. XLNet-Large was pre-trained on 159 GB of uncompressed text. The training data set includes the *BooksCorpus* and English Wikipedia, as with RoBERTa. Additionally, the news corpus *Giga5*[46], as well as portions of the *ClueWeb 2012-B*[12] and *Common Crawl* corpora, which contain various web texts, are part of the training data set. XLNet-Large is comparable in size to RoBERTa with 24 layers, a hidden size of 1024, 16 attention heads and 340M parameters.

When evaluated on the GLUE, RACE and SQuAD benchmarks, XLNet-Large outperformed RoBERTa slightly. What makes XLNet-Large additionally interesting for this thesis is that it has been benchmarked on several text classification data sets, including *AGNews* and *DBpedia*[72]. In these text classification tasks, XLNet-Large outperformed all competing models. These results suggest a potential good performance in classifying design decisions, which is why we apply and evaluate XLNet-Large as another pre-trained transformer language model.

BERTOverflow The third BERT-based model we experiment with is BERTOverflow, developed as part of a study by Tabassum et al.[58] on named entity recognition in software engineering tasks. BERTOverflow is a BERT-Base model trained on a domain-specific text corpus of 152M sentences collected from the programmer community platform StackOverflow. We estimate the data set size to be approximately 10 GB, assuming 70 bytes per sentence. BERTOverflow's transformer architecture includes 12 layers, a hidden size of 768, 12 attention heads, and 110M parameters.

BERTOverflow was developed with the intention of being better suited to handle language from the domain of software engineering or programming. However, it remains unclear whether BERTOverflow is superior in this regard compared to the other models. RoBERTa and XLNet were pre-trained on training data sets that are more than ten times larger, likely including numerous programming and software engineering-related texts as well. Therefore, we apply BERTOverflow to our approach to investigate how suitable it is for classifying design decisions compared to the other transformer LMs.

GPT-3 Lastly, we examine the capabilities of the large language model GPT-3 in classifying design decisions. GPT-3 is based on a decoder-only transformer architecture, unlike the BERT-based models, focusing on text generation. The model is trained to predict the next token in a given token sequence, based on all tokens *before* the token to be predicted. Hence, GPT-3 only considers the left context when processing a token and therefore is not classified as bidirectional. GPT-3 stands out from other language models due to its enormous scale in terms of the network size and the training corpus. GPT-3 was pre-trained with a data set of about 324GB in compressed size consisting primarily of filtered text from the CommonCrawl data set, complemented with some smaller text corpora. OpenAI trained GPT-3 models in different sizes. The largest model includes 96 layers, 96 attention heads, a hidden size of 12288, and a total of 175B trainable parameters, which is about 50x more than RoBERTa or XLNet. GPT-3 has been evaluated on various zero- and few-shot tasks, i.e. with only a few or no training samples, highlighting GPT-3's strength. To investigate the impact of size and ability to handle new NLP tasks with limited training samples on the classification of design decisions, we also include GPT-3 in our experiments.

OpenAI currently deploys GPT-3 as a commercial product in four different versions and prices. We opt for the model *Curie* as a good trade-off between size and pricing. According

	RoBERTa-Large	**XLNet-Large**	**BERTOverflow**	**GPT-3**
Architecture	Encoder	Encoder	Encoder	Decode
Parameters	355M	340M	110M	175B *(up to)*
Objective	Masked-LM	Permutation-LM	Masked-LM	Next Token Pred.
Corpus Size	161 GB	159 GB	~ 10 GB	324 GB *(compressed)*

Table 4.1: The pre-trained language models we employ and their main features.

to the description on the website, we assume that *Curie* is the second largest deployment of GPT-3. The exact hyperparameters of the model, however, are not publicly available. Table 4.1 summarises the language models employed and evaluated in the scope of this research.

4.2.2 Implementation

In the following, we provide insights into the implementation of the local and flat classifiers. The implementation of classifiers differs between GPT-3 and BERT-based models. This is primarily due to the inherent difference in the encoder-only or decoder-only architecture and the resulting property that BERT-based models output contextualised vectors while GPT-3 outputs tokens (sequences). Additionally, the practical difference is that we fine-tune GPT-3 through an API that limits our feasibility, while BERT-based models are freely accessible. We will first present the implementation of the BERT-based classifiers.

4.2.2.1 BERT-based Classifiers

The classifiers based on BERT models are designed as depicted in Figure 4.2. They consist of three key components: the language model specific tokenizer, the transformer LM itself, and a classification head.

The text classifier depicted in Figure4.2 processes an input text by first having it tokenized by the associated tokenizer. The tokens are then embedded by the transformer LM and processed through the stack of decoder layers to ultimately emit contextualised token embeddings. However, the individual token embeddings are not used any further and are thus ignored. Instead, we use the *pooled output* of the model. The *pooled output* is a contextualised vector representation of the entire token sequence. In XLNet and BERTOverflow, the pooled output is computed from the last hidden state of the [CLS] token, set for next-sentence prediction. Since RoBERTa does not implement next-sentence prediction, it uses its own <s> token for the pooled output. The pooled output now serves as the input to the classification head. The classification head essentially consists of a single dense linear layer that projects the 1024 or 768 input dimensions to the logits of the *n* class labels, determining the target classification.

For the implementation of the classifiers in Python, we relied on the open-source library Hugging Face [65]. Hugging Face provides an extensive repository of pre-trained transformer models, which have already been equipped with a classification head or other task-specific extensions. In addition, the library offers a variety of tools and interfaces that

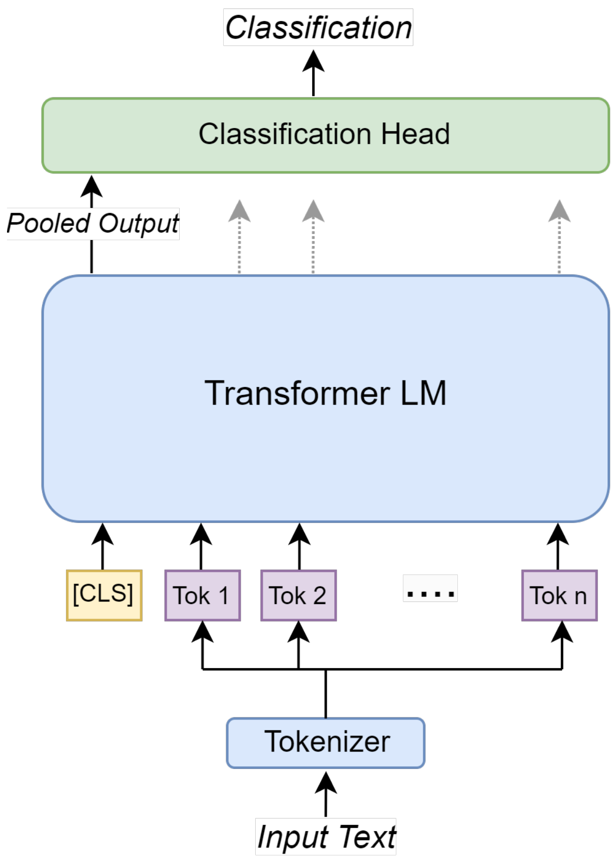

Figure 4.2: Design of a classifier with a BERT-based model.

aids the entire fine-tuning process For training the models, the training hyperparameters play an important role. To efficiently find the optimal hyperparameters, we perform an extensive hyperparameter search, considering the seed for network initialisation, the learning rate, the number of training epochs, the training batch size and the weight decay for the AdamW optimiser. The hyperparameter search is performed using a Bayesian optimiser, implemented with the Optuna framework, with the objective of maximising classification accuracy. The data set used for the hyperparameter search is split into 80% training and 20% validation data using stratified sampling and a fixed random seed, ensuring reproducibility

4.2.2.2 GPT-3 Classifier

Since GPT-3 is a decoder-only transformer model, the way it is used for text classification differs from the other models. The objective of GPT-3 is to predict the most likely next token of a token sequence. This feature can be leveraged for text classification by fine-tuning GPT-3 to predict the desired class label as the next token.

For fine-tuning, we use the API provided by OpenAI [19] and follow the recommendations outlined in their accompanying documentation. In order to fine-tune a GPT-3 model, we need to provide the API with the training data in the form of prompt-completion pairs. In our case, a prompt consists of the text line to be classified, concatenated with a separator string of the form "`\n\n==\n\n`". The separator string signals the model that the prompt ends and completion is expected. The completion, in our case, is the corresponding class name. We have reduced the class names to be represented as one token in order to reduce costs and minimize the risk of an out-of-bound prediction. For example, the class *Programming Language* is reduced to *programming*. Hence, a training sample in .json format might look like this:

```
{"prompt":"The application is written in Plankalkül\n\n==\n\n", "completion":"programming"}
```

The classification process is straightforward by sending a prompt, including the separator string, to the model and interpreting the predicted token, or completion, as the classification.

The API for fine-tuning a GPT-3 model offers the ability to set basic training hyperparameters such as batch size, number of epochs, and learning rate. Since the usage of the API is billed based on the number of processed tokens, we omit hyperparameter tuning for cost reasons and use the default hyperparameters provided by OpenAI for all classifiers.

4.3 Baseline

In order to better evaluate the actual benefit of transformer LMs for the classification of design decisions, we develop four baseline models for both the flat global and hierarchical LCPN classifiers, based on the machine learning algorithms naive Bayes, random forest, logistic regression, and support vector machine (SVM) as part of this work.

When developing the models, we apply various techniques to transform the raw texts into feature vectors. These techniques can be divided into three pipeline steps:

1. Pre-Processing The first step involves operations that clean up the text and remove noise as necessary. For this purpose, we consider the processing steps of lower-casing, removal of punctuation and special characters, replacing URLs with generic strings, stop-word removal and stemming.

2. Tokenization For tokenization, we use word tokenization, that divides the text into individual words, and character tokenization, dividing the text into individual characters.

3. Featurer Extraction & Selection For feature extraction, we use either single tokens or token n-grams. Depending on the tokenization, this results in single characters, single words, character n-grams or word n-grams. We also consider n-gram ranges for the n-grams. An n-gram range of, for instance, 2-4 means that 2-grams, 3-grams, and 4-grams are captured. For calculating the feature values, we use either binary count, absolute or relative count, or tf-idf. As these feature extraction methods can result in very high-dimensional and sparse feature vectors, we apply the feature selection methods Chi2, Mutual Information, or ANOVA F-Value by calculating the relevance of each feature using one of these methods and selecting the top n features. After this step, we obtain the final feature vectors, which are used to train the respective model.

Considering all these described techniques results in numerous possible combinations of pre-processing, tokenization, feature extraction, and feature selection. To find the optimal configuration, we performed Bayesian optimisation, analogue to the training hyperparameter search for the transformer LM approaches, using the Optuna framework. In addition, we included essential machine learning algorithm-specific hyperparameters to optimise them as well.

For the implementation of all baseline approaches, including all pipeline steps and the training of the naive Bayes, random forest, logistic regression and SVM models, we use the open-source library scikit-learn [52].

5 Evaluation

In the evaluation, we follow a GQM plan, as proposed by [5], shown in section 5.3. In order to obtain the most detailed information about which classification steps in the hierarchical approach perform well and which perform worse, we evaluate each local classifier individually. We measure the overall performance of each local classifier and the classification performance for each child class. Eventually, we evaluate the overall hierarchical global classifier and compare it to the flat global approaches.

5.1 Data Set

In this study, two data sets were used for training and evaluation purposes, each consisting of publicly available software architecture documentation (SADs) from open-source projects. Table 5.1 shows the data sets with the respective projects included. The SADs are structured in a way that each line represents a piece of text to be classified. Usually, one line contains one sentence.

The first data was used for cross-validation, as described in section 5.2. This data set originates from the publication of Keim et al.[33] and includes 17 open-source projects. Although the data set has already been labelled by the authors, we relabelled the entire data set manually. On the one hand, we wanted to gain a better understanding of the data set and of the design decision classes. On the other hand, we intended to ensure consistent labelling throughout both data sets, as the second data set was not pre-labelled. When labelling the data sets, we meticulously followed the class definitions of Keim et al. In some SAD lines. We always classified the first or foremost design decision in case several design decisions are expressed in one SAD line. For example, the sentence *"The application is built using Spring and deployed on an Apache Web Server"* contains both a design decision of the Framework class and one of the Platform class. In this case, we classified the decision as Framework, as it appears first in the sentence.

In addition to the first data set, a second data set consisting of 4 open-source projects, as also seen in Table 5.1, was used. The second data set had not been previously annotated and was manually labelled by us in the same manner described for the first data set. We held back the second data set exclusively as a test data set. Hence it was not employed for any training. We refer to this data set as the hold-back data set.

5.2 Validation Methods

For validation, we proceed by applying the GQM-plan for each transformer approach and each baseline. In order to measure the performance metrics listed in the GQM-plan, we

No.	Project	Domain	#Lines	Link
– Data Set 1 –				
1	ZenGarden	Media	109	`https://github.com/mhroth/ZenGarden`
2	SpringXD	Data Management	95	`https://github.com/spring-projects/spring-xd`
3	BIBINT	Science	22	`https://github.com/pebbie/BIBINT`
4	ROD	Data Management	119	`https://github.com/apohllo/rod`
5	tagm8vault	Media	16	`https://github.com/metafacets/tagm8-vault`
6	MunkeyIssues	Software Development	23	`https://github.com/seandgrimes/MunkeyIssues`
7	OnionRouting	Networking	51	`https://github.com/mangei/onion-routing`
8	Calipso	Web Development	30	`https://github.com/cliftonc/calipso`
9	IOSched	Event Management	81	`https://github.com/google/iosched`
10	MyTardis	Data Management	100	`https://github.com/mytardis/mytardis`
11	SCons	Software Development	79	`https://scons.org`
12	OpenRefine	Data Management	21	`https://github.com/johnconnelly75/OpenReïňĄne`
13	Beets	Media	125	`https://github.com/steinitzu/beets`
14	Teammates	Teaching	252	`https://github.com/TEAMMATES/teammates`
15	QMiner	Data Analysis	92	`https://github.com/qminer/qminer`
16	Spacewalk	Operating System	38	`https://github.com/spacewalkproject/spacewalk`
17	CoronaWarnApp	Healthcare	369	`https://github.com/corona-warn-app/cwa-documentation`
– Data Set 2 –				
1	BigBlueButton	Communication	85	`https://github.com/JabRef/jabref`
2	JabRef	Data Management	15	`https://github.com/bigbluebutton`
3	MediaStore	Media	37	`https://github.com/DescartesResearch/TeaStore`
4	TeaStore	E-Commerce	43	`https://sdq.kastel.kit.edu/wiki/Media_Store`

Table 5.1: Projects of the two data sets with their SAD's number of lines

used three different validation methods, two cross-validation methods and one with a fixed test data set.

Cross-Validation One of the two cross-validation methods is k-fold cross-validation with $k = 5$, i.e., 5-fold cross-validation. In 5-fold cross-validation, the data set is partitioned into five subsets or *"folds"* of equal size. In a test run, the model is trained on four folds while one is held back for testing. This process is repeated 5 times, so each fold has been used as a test set in exactly one iteration. Eventually, the average metrics across all five iterations determine the final performance metrics.

In addition to 5-fold cross-validation, we use group cross-validation, which we group by projects. We refer to this as project-fold or p-fold cross-validation. With p-fold cross-validation, we partition the data set in a way that each project appears in exactly one of the p partitions. Since the partitioning seeks to generate folds of approximately equal size, the individual folds may consist of a different number of projects. Therefore, the folds may have different numbers of projects. With p-fold cross-validation, we can measure the performance of the models on unseen projects. This is relevant because specific correlations in the data may be present within one project but not apply to other projects. Therefore, p-fold cross-validation allows us to better assess how well a classifier generalises.

For both cross-validation methods, we apply stratified sampling. This aims to ensure equal class distributions across all folds. With stratification, it is expected to obtain more balanced results with low variance during the testing rounds. Furthermore, it helps to better represent the characteristics and distribution of real-world data [35].

As mentioned already, for both cross-validation procedures, we apply a five-split, i.e. k=5 and p=5, respectively. Five and ten-folds are common configurations for cross-validation as they provide a good compromise between variance and bias of the individual test runs [74]. We eventually decided on a five-split because we are limited in computational and time costs due to a large number of classifiers we validate.

Validation on the Hold-Back Data Set Lastly, we validate all approaches on the hold-back data set, described in section 5.2, that is held back as a test data set. Hence the model is trained on the first data set and validated on the hold-back data set. The validation on the hold-back data set provides an exemplary performance measure on new, unseen projects without stratification and results being averaged over multiple test runs. Therefore, this validation most closely resembles a real application scenario.

5.3 GQM-Plan

In order to have a structured and reasonable proceeding for the evaluation, we have set up a GQM plan as proposed by [5]. The GQM plan includes an overarching goal and concrete questions for which metrics have been defined that aim to answer the questions measurably. The GQM plan is presented below.

Goal:
Assessing the classification performance of design decisions in software architecture documentation.

Q1: ***How well do the local classifiers in the hierarchical classifier predict, i.e., the child class x of the parent class X?***

In Q1, we consider a single local classifier of the LCPN approach that classifies the child classes x of the parent class X. With the metrics listed, we measure how well the classification of x works in a one-vs-the-rest fashion.

M1.1: precision p_x of class x

M1.2: recall r_x of class x

M1.3: f1-score $f1_x$ of class x

Q2: ***How well does the classifier generally classify design decisions of parent class X***

With $Q2$, we again analyse a single local classifier of parent class X, but measure the overall performance of the local classifier by aggregating and averaging the individual scores from Q1 with the metrics listed below.

M2.1: macro-averaged precision p_{mac}

M2.2: macro-averaged recall r_{mac}

M2.3: macro-averaged f1-score $f1_{mac}$

M2.4: accuracy A

M2.5: Matthews Correlation Coefficient MCC

Q3: ***How good is the final classification into the leaf node classes?***

With question Q3 we want to investigate how well the final tool classifies design decisions into all 25 leaf node classes. In the LCPN approach, this is the combination of all local classifiers, and in the global approach, the direct classification result.

M3.1: accuracy A

M3.2: micro-averaged precision p_{mac}

M3.3: micro-averaged recall r_{mac}

M3.4: micro-averaged f1-score $f1_{mac}$

M3.5: hierarchical precision p_H

M3.6: hierarchical recall r_H

M3.7: hierarchical f1-score $f1_H$

M3.8: Matthews Correlation Coefficient *MCC*

5.4 Evaluation Results

In this section, we present the evaluation results. We start by analysing the results of the local classifiers in subsection 5.4.1 in accordance with Q1 and Q2 of the GQM plan. Subsequently, we recapitulate the results and draw an interim conclusion of our findings in subsection 5.4.2. Finally, in subsection 5.4.3, we examine the evaluation results of the hierarchical global and flat global classifiers according to Q3.

5.4.1 Local Classifiers

In the following, we delve into the validation results of the local classifiers. For each local classifier, we present the results of all transformer-based approaches and the result of the best baseline approach comprehensively. In addition, we compare the measured performance between the three validation methods. The performance of each local classifier is determined by analysing the metrics for the classifier overall and the metrics for the classification performance of each child class.

5.4.1.1 Root

First, we look at the root classifiers, which identify *Design Decision*s by differentiating between the classes *Design Decision* and *No Design Decision.* The local root classifier can utilise the entire data set with a relatively balanced class distribution containing 65% of *Design Decisions.* Table 5.3 shows the class distribution for the corss validation data set (in front of the plus) and the hold-back data set (behind the plus). Therefore, we expected relatively good performances for this classifier. XLNet, and GPT-3 come closest to meeting these expectations, achieving an f1-score and accuracy of .90 in k-fold cross-validation and an MCC of .80. RoBERTa is just behind, as shown in Table 5.2. BERTOverflow, in contrast, showed inferior results and is barely better than the random forest. Between the three evaluation methods, evaluation performed slightly worse on unseen projects, except that BERTOverflow and RoBERTa achieved the highest scores on the hold-back projects. If we look at the performance of the individual classes in Table 5.3, we see that RoBERTa achieved a higher f1-score in classifying *Design Decision*, while all other models performed better on the *No Design Decision* class. In summary, we observe that all transformer models outperformed the best baseline, the random forest, with BERTOverflow inferior to the other transformer models. The performance of the root classifiers was relatively good compared to the other classifiers, whose evaluation is discussed subsequently.

Root		p	r	$f1$	A	MCC
RoBERTa	k	.90	.84	.86	**.90**	.73
	p	.85	.78	.81	**.87**	.62
	h	.94	.88	**.91**	**.94**	**.82**
XLNet	k	.91	.90	**.90**	**.90**	**.80**
	p	.87	.84	.85	**.87**	.71
	h	.91	.83	.87	.92	.74
B-Overflow	k	.85	.82	.83	.82	.67
	p	.81	.78	.77	.80	.57
	h	.88	.84	.86	.92	.72
GPT-3	k	.91	.90	**.90**	**.90**	**.80**
	p	.89	.87	**.88**	**.87**	**.76**
	h	.86	.76	.81	.89	.61
Baseline (RF)	k	.83	.80	.81	.80	.62
	p	.77	.76	.75	.76	.52
	h	.79	.77	.78	.78	.54

Table 5.2: Evaluation results of the root classifier

5.4.1.2 Design Decision

Next, we look at the local *Design Decision* classifiers, which classify into *Existence Decisions*, *Property Decisions*, and *Executive Decisions*. We observe a lower overall performance than with the root classifiers. Comparing the language models shows that RoBERTa, XLNet, and GPT-3 achieved comparable results and again outperformed BERTOverflow, which could not beat the baseline. When evaluated on unseen projects, RoBERTa stood out from the other models, whose performance dropped significantly more compared to k-fold cross-validation. The evaluation of the performance per class shows that the poor classification of the*Property Decision* classes had a pronounced negative impact on the overall result. The hold-back projects data set contains only four *Property Decisions*. Based on the recall, we see that XLNet and RoBERTa classified two of the four samples as such. All other models did not identify the *Property Decisions*. Further, observe that the results correlate with the distribution of classes. Accordingly, the classification of *Existence Decisions* performed best, and *Property Decisions* performed worst. Overall, the evaluation of the *Design Decision* classifiers shows that *Property Decision* failed to be classified in our approaches. Furthermore, it shows susceptibility to uneven class sizes. When comparing the approaches, BERTOverflow was not better than the baseline, while the other transformer models were superior.

Root		**Design Dec.**			**No Design Dec.**		
		p	r	$f1$	p	r	$f1$
RoBERTa	k	.90	.97	**.93**	.89	.72	.79
	p	.88	.95	**.91**	.82	.62	.70
	h	.93	.77	**.84**	.95	.99	**.97**
XLNet	k	.91	.84	.87	.90	.95	**.92**
	p	.87	.78	.82	.87	.90	.88
	h	.83	.71	.77	.93	.97	.95
B-Overfl.	k	.87	.71	.78	.83	.93	.88
	p	.84	.60	.70	.78	.93	.85
	h	.83	.71	.77	.93	.97	.95
GPT-3	k	.92	.84	.87	.90	.95	**.92**
	p	.90	.80	.84	.88	.94	**.91**
	h	.83	.54	.66	.90	.97	.93
Baseline (RF)	k	.84	.67	.75	.81	.92	.86
	p	.73	.67	.69	.81	.84	.81
	h	.76	.69	.72	.81	.85	.83
Class Distrib.		*984 + 143*			*638 + 35*		

Table 5.3: Evaluation results for identifying design decision

Design Dec.		p	r	$f1$	A	MCC
RoBERTa	k	.77	.68	.71	.83	.62
	p	.60	.56	**.56**	**.81**	**.55**
	h	.67	.71	.69	.85	**.56**
XLNet	k	.80	.72	**.75**	**.85**	**.66**
	p	.48	.46	.45	.76	.36
	h	.73	.68	**.71**	**.86**	.54
B-Overflow	k	.64	.55	.57	.79	.48
	p	.47	.39	.37	.73	.23
	h	.27	.33	.30	.82	.00
GPT-3	k	.79	.70	.74	.84	.63
	p	.49	.50	.50	.78	.46
	h	.48	.52	.50	.82	.44
Baseline (NB)	k	.61	.60	.60	.74	.45
	p	.43	.44	.42	.67	.26
	h	.27	.33	.30	.82	.00

Table 5.4: Evaluation results of the *design decision* classifier

Design Dec.		Existence			Property			Executive		
		p	*r*	*f*1	*p*	*r*	*f*1	*p*	*r*	*f*1
RoBERTa	*k*	.87	.92	.89	.69	.44	.53	.74	.70	.72
	p	.83	.88	**.85**	.25	.12	**.15**	.73	.68	**.69**
	h	.93	.89	**.91**	.50	.50	.50	.59	.73	**.65**
XLNet	*k*	.88	.93	**.90**	.69	.51	.57	.83	.72	**.77**
	p	.76	.91	.82	.11	.02	.03	.58	.46	.51
	h	.91	.91	**.91**	.67	.50	**.57**	.61	.64	.62
B-Overfl.	*k*	.82	.93	.87	.39	.19	.25	.70	.53	.60
	p	.75	.81	.73	.00	.00	.00	.66	.36	.38
	h	.82	1.0	.90	.00	.00	.00	.00	.00	.00
GPT-3	*k*	.87	.93	**.90**	.73	.48	**.58**	.75	.68	.72
	p	.80	.87	.83	.00	.00	.00	.68	.65	.65
	h	.90	.87	.89	.00	.00	.00	.54	.68	.60
Baseline (NB)	*k*	.85	.82	.83	.41	.34	.37	.57	.64	.60
	p	.75	.77	.75	.07	.01	.02	.47	.53	.49
	h	.82	1.0	.90	.00	.00	.00	.00	.00	.00
Class Distrib.		*689 + 117*			*73 + 4*			*222 + 22*		

Table 5.5: Evaluation results for the classification of *design decision* child classes

5.4.1.3 Existence Decision

When classifying *Existence Decisions* into *Structural Decisions*, *Arrangement Decisions*, and *Behavioural Decisions*, we observe that RoBERTa, XLNet and GPT3 gave the best results, with an accuracy of around .80, a macro averaged f1-score of around .70 and an MCC of roughly .65. We again find that BERTOverflow underperformed here and barely exceeded the random forest baseline. When we compare the results of the evaluation methods, we see that the hold-back projects and the p-fold cross-validation provided slightly worse results, which is in line with expectations. On the p-fold cross-validation, GPT-3 and RoBERTa delivered the best results. On the hold-back projects, XLNet was the most successful. Analysing the metrics per class, the results reveal a significant disparity in the classification performance between the *Arrangement Decision* class and the other classes. This is most apparent in the results of BERTOverflow, which achieved f1 scores close to 0. The other transformer models showed a decrease of about .20 - .30 in their f1 scores when classifying samples into the arrangement class compared to the other two classes. Furthermore, the *Structural Decision* achieved slightly lower results than the *Behavioural Decision*, which is again most pronounced at BERTOveflow. It is likely that the disparate classification performance is attributed to the unequal distribution of classes, with the arrangement decision being particularly underrepresented. This had the highest impact on BERTOverflow, suggesting that RoBERTa, XLNet and GPT-3 are more resistant to class imbalance.

Overall, the evaluation of the *Existence Decision* classifier again shows BERTOverflow to be outperformed by other transformer models. Additionally, the performance was affected by class imbalance, most notably seen in BERTOverflow and the baseline.

Existence Dec.		*p*	*r*	*f*1	*A*	*MCC*
RoBERTa	*k*	.77	.70	.72	**.83**	**.69**
	p	.76	.64	**.66**	**.76**	.52
	h	.84	.58	.69	.76	.55
XLNet	*k*	.86	.69	.72	.82	.66
	p	.66	.67	.65	.74	.49
	h	.85	.65	**.74**	**.78**	**.59**
B-Overflow	*k*	.56	.53	.53	.74	.50
	p	.45	.43	.40	.61	.30
	h	.45	.46	.45	.65	.37
GPT-3	*k*	.83	.69	**.75**	.81	.64
	p	.66	.64	.65	.74	**.53**
	h	.66	.61	.63	.73	.49
Baseline (RF)	*k*	.62	.52	.53	.73	.47
	k	.46	.44	.42	.58	.26
	k	.55	.42	.45	.62	.32

Table 5.6: Evaluation results of the *existence decision* classifier

5.4.1.4 Property Decision

Next, we evaluate the local classifiers for the *Property Decisions*, which distinguish design decisions into guideline and design rule. We see the naive Bayes approach gave the best overall result for the k-fold cross-validation but poor results on unseen projects i.e. for the p-fold cross-validation and hold-back projects. For the p-fold cross-validation, we got the best results with BERTOverflow. The result with the hold-back projects always yielded an MCC of 0 since no samples of the class *Guideline* are included in the test data set. Nevertheless, we can see that GPT-3 and BERTOverflow correctly identified all four *Design Rules.* For all approaches, we observe a low MCC, despite moderate to high accuracies. The evaluation of the performances per class indicates that this low MCC is primarily due to the poor performance in predicting the *Guideline* class, which is significantly lower than that for the *Design Rule* class. Interestingly, in p-fold cross-validation, BERTOverflow was the only model that achieved better performance for the *Guideline* but poor performance for the *Design Rule.* Evaluating the *Property Decision* classifier, we find that all classifiers yielded low MCCs and moderate to low macro-averaged f1-scores. Mostly, this is due to deficient performance in classifying *Guidelines.* Again, the class imbalance problem in the data set could contribute to poor performance. However, another factor may be that *design rule* and *Guideline* are very similar concepts, differing only because the *Guideline* is

Existence Dec.		Structural			Arrangement			Behavioural		
		p	*r*	*f*1	*p*	*r*	*f*1	*p*	*r*	*f*1
RoBERTa	*k*	.84	.78	**.80**	.63	.43	.51	.85	.90	**.87**
	p	.73	.68	**.69**	.80	.37	.48	.76	.86	**.80**
	h	.79	.78	**.78**	.17	.29	**.55**	.73	.81	.77
XLNet	*k*	.81	.78	**.80**	.93	.39	.51	.83	90	.86
	p	.74	.65	**.69**	.53	.52	.50	.72	.83	.77
	h	.82	.72	.77	1.0	.33	.50	.73	.89	**.80**
B-Overfl.	*k*	.72	.68	.69	.20	.07	.10	.77	.84	.80
	p	.54	.38	.43	.20	.03	.06	.61	.87	.71
	h	.73	.50	.59	.00	.00	.00	.61	.89	.72
GPT-3	*k*	.81	.74	.77	.85	.44	**.58**	.82	.90	.86
	p	.60	.55	.57	.68	.50	**.56**	.71	.88	.77
	h	.75	.74	.75	.50	.33	.40	.71	.75	.73
Baseline (RF)	*k*	.70	.65	.67	.40	.07	.11	.75	.84	.79
	p	.47	.40	.42	.30	.12	.17	.60	.80	.67
	h	.66	.45	.53	.35	.09	.14	.63	.71	.67
Class Distrib.		*271 + 58*			*32 + 6*			*386 + 53*		

Table 5.7: Evaluation results for the classification of *existence decision* child classes

not a hard rule and thus not mandatory. Therefore, the classes *Guideline* and *Design Rule* are sometimes not clear-cut.

5.4.1.5 Executive Decision

In our evaluation of the *Executive Decision* classifiers, none of the approaches achieved an MCC greater than 0. With this classifier, we face a severe class imbalance with 218 *Technological* classes and only 4 *Organisational/Process-related* classes. Looking at the class-specific performances, we see that for cross-validation, none of the approaches correctly classified a *organisational/Process-related* design decision. For the hold-back projects, only 1 out of 22 design decisions belong to the class *Organisational/Process-related* related. Here, XLNet classified flawless, while the other models did not correctly classify the one class. Overall, the data set appears to be insufficient for the local *Executive Decision* classifier, as the dearth of the *organisational/Process-related* class led to unreliable overall predictions, making it hard to assess the approaches.

5.4.1.6 Structural Decision

In evaluating the *Structural Decision* classifier, RoBERTa, XLNet, and GPT-3 showed similar results in both cross-validation methods. BERTOverflow performed slightly worse than these models but still better than the SVM baseline. We further observe that all models showed a significant decrease in performance with the p-fold cross-validation

Property Dec.		p	r	$f1$	A	MCC
	k	.62	.62	.61	.89	.27
RoBERTa	p	.53	.53	.51	**.87**	.11
	h	.00	.00	.00	.00	.00
	k	.75	.63	.65	.90	.36
XLNet	p	.43	.45	.43	.76	.00
	h	1.0	.25	.40	.25	.00
	k	.60	.66	.59	.70	.25
B-Overflow	p	.58	.63	**.54**	.76	**.30**
	h	1.0	1.0	**1.0**	**1.0**	.00
	k	.63	.73	.68	.90	.39
GPT-3	p	.63	.63	.40	.63	.25
	h	1.0	1.0	**1.0**	**1.0**	.00
	k	.76	.69	**.71**	**.91**	**.45**
Baseline (NB)	p	.25	.32	.25	.40	.02
	h	.00	.00	.00	.00	.00

Table 5.8: Evaluation results for the classification of *property decision* child classes

Property Dec.		**Guideline**			**Design Rule**		
		p	r	$f1$	p	r	$f1$
	k	.31	.30	.28	.93	.95	.93
RoBERTa	p	.40	.27	.30	.67	.80	**.73**
	h	-	-	-	.00	.00	.00
	k	.60	.25	.35	.90	.10	.15
XLNet	p	.20	.10	.13	.66	.80	**.73**
	h	-	-	-	1.0	0.25	.40
	k	.28	.60	.38	.91	.71	.80
B-Overfl.	p	.81	.87	.72	.34	.40	.37
	h	-	-	-	1.0	1.0	**1.0**
	k	.33	.50	**.40**	.93	.96	.94
GPT-3	p	.25	1.0	**.40**	1.0	.25	.40
	h	-	-	-	1.0	1.0	**1.0**
	k	.60	.40	.47	.92	.98	**.95**
Baseline (NB)	p	.01	.13	.03	.48	.50	.47
	h	-	-	-	.00	.00	.00
Class Distrib.		*10 + 0*			*63 + 4*		

Table 5.9: Evaluation results of the *property decision* classifier

Executive Dec.		*p*	*r*	*f*1	*A*	*MCC*
RoBERTa	*k*	.49	.50	**.50**	**.98**	.00
	p	.49	.50	**.50**	**.98**	.00
	h	.48	.50	.49	.95	.00
XLNet	*k*	.49	.50	**.50**	**.98**	.00
	p	.49	.50	**.50**	**.98**	.00
	h	1.0	1.0	**1.0**	**1.0**	**1.0**
B-Overflow	*k*	.49	.50	**.50**	**.98**	.00
	p	.48	.50	.49	.96	.00
	h	.48	.50	.49	.95	.00
GPT-3	*k*	.49	.50	.49	**.98**	.00
	p	.46	.50	.48	.93	.00
	h	.48	.50	.49	.95	.00
Baseline (RF)	*k*	.50	.50	**.50**	**.98**	.00
	k	.50	.50	**.50**	**.98**	.00
	h	.48	.50	.49	.95	.00

Table 5.10: Evaluation results of the *executive decision* classifier

Executive Dec.		**Technological**			**Orga./Process.**		
		p	*r*	*f*1	*p*	*r*	*f*1
RoBERTa	*k*	.98	1.0	**.99**	.00	.00	.00
	p	.98	1.0	**.99**	.00	.00	.00
	h	.95	1.0	.98	.00	.00	.00
XLNet	*k*	.98	1.0	**.99**	.00	.00	.00
	p	.98	1.0	**.99**	.00	.00	.00
	h	1.0	1.0	**1.0**	1.0	1.0	1.0
B-Overfl.	*k*	.98	1.0	**.99**	.00	.00	.00
	p	.96	1.0	.98	.00	.00	.00
	h	.95	1.0	.98	.00	.00	.00
GPT-3	*k*	.98	1.0	**.99**	.00	.00	.00
	p	.93	1.0	.96	.00	.00	.00
	h	.95	1.0	.98	.00	.00	.00
Baseline (RF)	*k*	.98	1.0	**.99**	.00	.00	.00
	p	.96	1.0	.98	.00	.00	.00
	h	.95	1.0	.98	.00	.00	.00
Class Distrib.		*218 + 21*			*4 + 1*		

Table 5.11: Evaluation results for the classification of *executive decision* child classes

compared to the k-fold cross-validation. The results of the hold-back projects varied between all approaches, especially with respect to the MCC. If we look at the classification performances of the individual classes, we see that the performance of *Intra-Systemic* was significantly better than the performance of *Extra-Systemic* throughout all approaches. This is likely again a result of the class imbalance. To sum up, the results demonstrate that all transformer models outperform the baseline on this local classifier. Furthermore, a relatively large performance decline is observed during p-fold cross-validation. The results on hold-back projects are highly varied among the different approaches, with MCCs ranging from .28 to .70. Ultimately, GPT-3 emerges as the most suitable option, exhibiting the best performance on unseen projects and marginally lower performance than the best model during k-fold cross-validation.

Structural Dec.		p	r	$f1$	A	MCC
RoBERTa	k	.92	.82	**.85**	**.93**	**.73**
	p	.82	.70	.72	**.90**	**.49**
	h	.87	.56	.68	.74	.30
XLNet	k	.90	.82	**.85**	.92	.72
	p	.81	.65	.66	.88	.41
	h	.91	.74	.81	.84	.62
B-Overflow	k	.89	.77	.81	.90	.64
	p	.77	.69	.70	.89	.45
	h	.87	.56	.68	.74	.30
GPT-3	k	.96	.77	**.85**	.92	.69
	p	.73	.76	**.75**	.83	**.49**
	h	.89	.81	.85	.88	**.70**
Baseline (SVM)	k	.86	.71	.75	.89	.55
	p	.65	.60	.57	.78	.23
	h	.75	.67	.56	.80	.28

Table 5.12: Evaluation results of the *structural decision* classifier

5.4.1.7 Arrangement Decision

Next, we examine the evaluation results of the *Arrangement Decision* classifiers. Here we see that BERTOverflow outperformed the other transformer models in both cross-validation evaluation methods. In cross-validation, RoBERTa, XLNet, and GPT-3 are inferior to the baseline. Looking at the individual classification results of the three classes we see that BERTOverflow predicted the class *Architectural Style* very reliably but also predicted *Architectural Pattern* significantly better than the other approaches, especially compared to RoBERTa and XLNet. For all models, we observe that *Architectural Style* was the best classified class and *Reference Architecture* class the worst. When evaluating the hold-back projects, we see strongly differing results between the different models, with

Structural Dec.		**Extra-Systemic**			**Intra-Systemic**		
		p	*r*	*f*1	*p*	*r*	*f*1
RoBERTa	*k*	.90	.66	**.75**	.94	.98	**.96**
	p	.71	.55	.59	.94	.84	.84
	h	1.0	.12	.21	.73	1.0	.85
XLNet	*k*	.88	.68	**.75**	.93	.97	.95
	p	.70	.45	.49	.92	.84	.83
	h	1.0	.47	.64	.82	1.0	.90
B-Overfl.	*k*	.87	.56	.67	.91	.98	.94
	p	.61	.55	.57	.94	.83	.84
	h	.92	.65	**.76**	.87	.98	**.92**
GPT-3	*k*	1.0	.54	.67	.91	1.0	.95
	p	.83	.61	**.70**	.79	.99	**.86**
	h	.92	.65	**.76**	.89	.98	**.92**
Baseline (SVM)	*k*	.83	.45	.47	.89	.98	.94
	p	.54	.25	.31	.77	.95	.82
	h	.70	.14	.23	.80	.99	.89
Class Distrib.		*47 + 17*			*224 + 41*		

Table 5.13: Evaluation results for the classification of *structural decision* child classes

XLNet classified perfectly, while GPT-3 and the baseline yielded the worst possible results. The data subset for the *Arrangement Decision* classifier only consists of 32 samples. This is likely a major limitation to the classification performance of the models, especially for the *Reference Architecture* class, which only appears four times in the data set. Furthermore, it should be considered that a small data set size limit also the quality of the performance estimation.

5.4.1.8 Behavioral Decision

Subsequently, we examine the results of the evaluation of the *Behavioural Decision* classifier, which classifies into the classes *Relation*, *Function*, *Algorithm* and *Messaging*. Looking at the general evaluation results, RoBERTa, XLNet and GPT-3 achieved an MCC of over .70 in k-fold cross-validation, an accuracy of well over .80 and an f1-score of around .80. BERTOverflow performed poorer in k-fold cross-validation, with .35 percentage points lower MCC than RoBERTa and .26 percentage points lower than the baseline. Considering all the evaluation results, we see GPT-3 as the best-performing classifier for *Behavioral Decisions*. A striking observation is that all approaches scored significantly worse in the p-fold validation. For RoBERTa and XLNet, in particular, this is a difference of about .30 percentage points in both the MCC and the f1-score. BERTOverflow shows the least performance drop in p-fold cross-validation compared to k-fold. The discrepancy between k-fold cross-validation and the two validation methods on unseen projects is the smallest

Arrangement		p	r	$f1$	A	MCC
	k	.57	.59	.57	.77	.58
RoBERTa	p	.23	.33	.26	.68	.00
	h	.33	.50	.40	.67	.00
	k	.50	.53	.49	.71	.52
XLNet	p	.40	.44	.42	.77	.30
	h	1.0	1.0	1.0	1.0	**1.0**
	k	.74	.74	**.73**	.77	**.80**
B-Overflow	p	.51	.57	**.53**	**.78**	**.55**
	h	.90	.75	.82	.83	.63
	k	.53	.57	.55	.80	.65
GPT-3	p	.42	.36	.36	.53	.24
	h	.00	.00	.00	.00	.00
	k	.55	.62	.58	**.82**	.69
Baseline (LR)	p	.43	.51	.43	.64	.28
	h	.00	.00	.00	.00	.00

Table 5.14: Evaluation results of the *arrangement decision* classifier

Arrangement		**Arc. Style**			**Arc. Pattern**			**Reference Arc.**		
		p	r	$f1$	p	r	$f1$	p	r	$f1$
	k	.83	.93	.87	.69	.63	.64	.20	.20	.20
RoBERTa	p	.51	.60	.55	.00	.00	.00	.17	.40	**.23**
	h	-	-	-	.00	.00	.00	0.67	1.0	.80
	k	.83	.70	.72	.55	.70	.61	.10	20	.13
XLNet	p	.67	.80	.72	.20	.20	.20	.33	.33	.33
	h	-	-	-	1.0	1.0	**1.0**	1.0	1.0	**1.0**
	k	.88	1.0	**.93**	.93	.83	**.87**	.40	.40	**.40**
B-Overfl.	p	.81	.97	**.87**	.70	.73	**.72**	.00	.00	.00
	h	-	-	-	1.0	0.5	.67	.80	1.0	.89
	k	.77	1.0	.87	.83	.72	.77	.00	.00	.00
GPT-3	p	.70	.75	.67	.67	.39	.49	.00	.00	.00
	h	-	-	-	.00	.00	.00	.00	.00	.00
	k	.82	.95	.88	.83	.90	.86	.00	.00	.00
Baseline (LR)	p	.63	.78	.64	.66	.75	.66	.00	.00	.00
	h	-	-	-	.00	.00	.00	.00	.00	.00
Class Distrib.		*16 + 0*			*12 + 4*			*4 + 2*		

Table 5.15: Evaluation results for the classification of *arrangement decision* child classes

for the classification of function. This is likely contributed by the fact that this class, with 264 samples, is most frequently represented in the entire data set, and therefore the classifiers generalise better.

Behavioural		*p*	*r*	*f*1	*A*	*MCC*
RoBERTa	*k*	.85	.80	**.81**	**.88**	**.77**
	p	.60	.55	**.53**	.67	.47
	h	.85	.62	.72	.77	.63
XLNet	*k*	.84	.77	.79	.87	.73
	p	.57	.46	.47	.67	.43
	h	.78	.76	.77	**.81**	**.70**
B-Overflow	*k*	.63	.46	.49	.76	.43
	p	.43	.39	.39	.70	.37
	h	.74	.53	.62	.68	.44
GPT-3	*k*	.87	.76	**.81**	**.88**	.75
	p	.53	.53	**.53**	**.71**	**.59**
	h	.94	.74	**.83**	**.81**	**.70**
Baseline (LR)	*k*	.55	.62	.58	.82	.69
	p	.43	.51	.43	.64	.28
	h	.59	.27	.33	.66	.25

Table 5.16: Evaluation results for the classification of *behavioural decision* child classes

5.4.1.9 Extra-Systemic

We proceed to study the evaluation results for classifying *Extra-Systemic* design decisions. Comparing the transformer models with the naive Bayes baseline, we see that the baseline performed almost as well as RoBERTa, XLNet and GPT-3 in k-fold cross-validation and even better than BERTOverflow. However, the advantage of the transformers becomes apparent when evaluating unseen projects, i.e. p-fold cross-validation and hold-back projects. We see that BERTOverflow exhibits a significantly lower performance compared to the other transformer models. GPT-3 achieved the best overall results in the cross-validation, while XLNet clearly outperformed the other approaches on the hold-back projects. The evaluation of the performances of the individual classes indicates that BERTOverflow failed in the classification of data files, which led to overall poor performance. Furthermore, we note that in RoBERTa, XLNet and GPT-3, the *Integration* class was classified more reliably in the k-fold cross-validation and on the hold-back classes. In contrast, in p-fold cross-validation, better results were obtained for the *Data File* class. Therefore, we do not see a clear tendency which of the two classes is better classified.

Behavioural		**Relation**			**Function**			**Algorithm**			**Messaging**		
		p	*r*	*f*1	*p*	*r*	*f*1	*p*	*r*	*f*1	*p*	*r*	*f*1
RoBERTa	*k*	.82	.82	**.80**	.92	.95	**.93**	.85	.65	**.72**	.79	.79	.78
	p	.46	.25	.31	.71	.93	**.78**	.60	.44	**.48**	.65	.58	.57
	h	1.0	.20	.33	.73	1.0	.85	1.0	.78	**.88**	.67	.50	.57
XLNet	*k*	.87	.75	**.80**	.89	.94	.92	.76	.69	.70	.83	.71	.76
	p	.68	.24	**.33**	.67	.86	.74	.48	.36	.38	.46	.40	.41
	h	.75	.30	**.43**	.85	.97	.91	1.0	.78	**.88**	.50	1.0	.67
B-Overfl.	*k*	.83	.42	.54	.77	.97	.86	.43	.28	.32	.50	.16	.23
	p	.30	.26	.28	.70	.89	.78	.11	.06	.08	.62	.36	.42
	h	.67	.20	.31	.67	.97	.79	1.0	0.22	.36	.60	.75	.67
GPT-3	*k*	.82	.83	.79	.90	.96	**.93**	.75	.44	.54	1.0	.81	**.89**
	p	.40	.13	.20	.67	.91	.77	.44	.36	.40	.63	.71	**.59**
	h	1.0	.20	.33	.75	1.0	**.86**	1.0	.78	**.88**	1.0	1.0	**1.0**
Baseline (LR)	*k*	.44	.38	.40	.75	.83	.79	.25	.20	.25	.37	.32	.34
	p	.12	.04	.06	.62	.76	.67	.22	.10	.13	.38	.26	.27
	h	.26	.08	.12	.55	.64	.59	.90	.22	.35	.67	.15	.25
Class Distrib.		*45 + 10*			*264 + 10*			*40 + 9*			*37 + 4*		

Table 5.17: Evaluation results of the *behavioural decision* classifier

Extra-Systemic		*p*	*r*	*f*1	*A*	*MCC*
RoBERTa	*k*	.97	.93	.94	.96	.89
	p	.53	.58	.51	.80	.13
	h	.81	.57	.67	.65	.30
XLNet	*k*	.91	.84	.85	.89	.74
	p	.63	.59	**.60**	**.87**	**.78**
	h	.94	.95	**.94**	**.94**	**.89**
B-Overflow	*k*	.58	.57	.53	.77	.20
	p	.62	.61	.61	.74	.23
	h	.81	.57	.67	.65	.30
GPT-3	*k*	.98	.94	**.96**	**.97**	**.92**
	p	.63	.59	**.60**	**.87**	**.78**
	h	.81	.57	.67	.65	.30
Baseline (NB)	*k*	.95	.87	.88	.92	.81
	p	.40	.46	.41	.65	.11
	h	.67	.52	.49	.71	.26

Table 5.18: Evaluation results of the *extra-systemic* classifier

Extra-Systemic		**Data File**			**Integration**		
		p	r	$f1$	p	r	$f1$
RoBERTa	k	1.0	.85	.91	.94	1.0	.97
	p	.73	.97	**.79**	.33	.19	.23
	h	1.0	.14	.25	.63	1.0	.77
XLNet	k	.93	.72	.77	.89	.97	.93
	p	.74	.75	.74	.52	.42	.46
	h	.88	1.0	**.93**	1.0	0.9	**.95**
B-Overfl.	k	.40	.13	.20	.76	1.0	.86
	p	.50	.44	.46	.74	.78	**.76**
	h	1.0	.14	.25	.63	1.0	.77
GPT-3	k	1.0	.89	**.93**	.96	1.0	**.98**
	p	.74	.75	.74	.52	.42	.46
	h	1.0	.14	.25	.63	1.0	.77
Baseline (NB)	k	1.0	.73	.82	.90	.1.0	.95
	p	.07	.02	.01	.74	.71	.71
	h	.70	.14	.23	.63	.90	.74
Class Distrib.		*13 + 7*			*34 + 10*		

Table 5.19: Evaluation results for the classification of *extra-systemic* child classes

5.4.1.10 Intra-Systemic

In analyzing the results of the local I*ntra-Systemic* classifier, we observed that all approaches exhibited relatively high MCC and accuracy scores in k-fold cross-validation, with BERTOverflow performing the lowest. However, GPT-3 emerged as the best-performing model in this evaluation method. It is noteworthy that the classification performance was significantly poorer for the p-fold cross-validation, where BERTOverflow demonstrated the lowest MCC delta to the p-fold cross-validation and hence performed the best. For the hold-back projects, all models except GPT-3 achieved an MCC of 0, indicating the absence of correlation between their predictions and actual class assignments. Examining the distribution of the three classes, we see that 5 of 245 samples belonged to the *Interface* class. This is reflected in its poor classification performance, as the class was either rarely or never predicted correctly.

5.4.1.11 Class-Related

Next, we review the evaluation results of the *Class-Related* classifier, which classifies *Class-Related* design decisions into *Association*, *Class* and *Inheritance* Looking first at the cross-validations, we see that XLNet and GPT-3 performed best in k-fold cross-validation, and GPT-3 outperformed the other approaches in p-fold cross-validation. The approach with BERTOverflow gave the lowest results compared to the other transformers and even lower results than the baseline in the k-fold cross-validation. We find that RoBERTa, XLNet

Intra-Systemic		*p*	*r*	*f*1	*A*	*MCC*
RoBERTa	*k*	.62	.64	.63	.93	.87
	p	.27	.34	.25	.52	.05
	h	.10	.50	.16	.20	.00
XLNet	*k*	.62	.64	.63	.93	.86
	p	.40	.40	.34	.65	.28
	h	.52	.49	.51	.71	.00
B-Overflow	*k*	.58	.58	.58	.86	.73
	p	.53	.48	**.47**	**.77**	**.57**
	h	.57	.34	.43	.24	.00
GPT-3	*k*	.75	.75	**.75**	**.94**	**.89**
	p	.51	.48	.49	.71	.46
	h	.73	.75	.74	.83	.48
Baseline (SVM)	*k*	.59	.60	.60	.88	.77
	p	.48	.48	.45	.71	.43
	h	.48	.31	.29	.31	.00

Table 5.20: Evaluation results of the *intra-systemic* classifier

Intra-Systemic		**Interface**			**Component**			**Class-Related**		
		p	*r*	*f*1	*p*	*r*	*f*1	*p*	*r*	*f*1
RoBERTa	*k*	.00	.00	.00	.93	.95	**.94**	.94	.96	.95
	p	.00	.00	.00	.51	.96	.64	.30	.07	.11
	h	-	-	-	.20	1.0	.33	.00	.00	.00
XLNet	*k*	.00	.00	.00	.91	.96	.93	.94	.94	.94
	p	.20	.20	.20	.66	.98	**.76**	.34	.03	.06
	h	-	-	-	.25	.13	.17	.80	.85	.82
B-Overfl.	*k*	.00	.00	.00	.89	.84	.86	.84	.92	.88
	p	.00	.00	.00	.86	.59	.63	.75	.84	**.78**
	h	-	-	-	.14	.50	.22	1.0	.18	.31
GPT-3	*k*	.33	.33	**.33**	.96	.92	**.94**	.94	.99	**.96**
	p	.00	.00	.00	.78	.75	.75	.75	.68	.67
	h	-	-	-	.59	.63	**.59**	.91	.88	**.89**
Baseline (SVM)	*k*	.00	.00	.00	.88	.89	.88	.88	.91	.89
	p	.00	.00	.00	.74	.57	.61	.69	.87	.76
	h	-	-	-	.16	.40	.23	.80	.22	.35
Class Distrib.		*5 + 0*			*97 + 33*			*122 + 8*		

Table 5.21: Evaluation results for the classification of *intra-systemic* child classes

and GPT-3 generalised better than the baseline, especially on unseen projects, which can be seen from the fact that the MCC, the macro averaged f1-score, and the accuracy dropped significantly less in the p-fold cross-validation compared to the k-fold cross-validation. However, this does not apply to BERTOverflow. In the evaluation with the hold-back projects, we get identical results for all approaches. The evaluation of the individual class performance reveals that the primary advantage of GPT-3 lies in its classification of the *Inheritance* class, resulting in a higher overall f1-score compared to other approaches, despite lower accuracy in some instances. All other approaches showed low reliability in predicting the *Inheritance* class. The low representation of the *Inheritance* class in the data set is a potential factor contributing to its poor classification performance. In conclusion, RoBERTa, XLNet and GPT-3 outperform the baseline and BERTOverflow. Among the transformer models, GPT-3 generalises best to unseen projects, making it the leading approach. The overall performance of this classifier was mediocre. In our evaluation, the classification performance was likely limited due to the small number of training samples per class and the class imbalance regarding the *Inheritance* class.

Class-Related		*p*	*r*	*f*1	*A*	*MCC*
RoBERTa	*k*	.67	.62	.63	.79	.62
	p	.62	.57	.55	.80	.43
	h	.83	.75	.79	.75	.58
XLNet	*k*	.76	.71	.72	**.82**	**.69**
	p	.67	.57	.59	**.81**	.41
	h	.83	.75	.79	.75	**.58**
B-Overflow	*k*	.59	.56	.55	.71	.48
	p	.41	.41	.35	.69	.16
	h	.83	.75	.79	.75	**.58**
GPT-3	*k*	.90	.75	**.81**	.81	.67
	p	.78	.79	**.78**	.76	**.54**
	h	.83	.75	.79	.75	**.58**
Baseline (LR)	*k*	.73	.63	.65	.75	.54
	p	.47	.43	.42	.63	.10
	h	.83	.75	.79	.75	**.58**

Table 5.22: Evaluation results of the *class-related* classifier

5.4.1.12 Technological

The classifier for the *Technological* design decisions has the largest class output space with five child classes. All classes are represented in similar proportions in the data set, except for the tool class, which constitutes a much smaller proportion with only four samples. The comparison of the overall performances of the approaches makes GPT-3 stand out clearly. In all three evaluation methods, GPT-3 performed best and showed the highest values in

Class-Related		**Association**			**Class**			**Inheritance**		
		p	*r*	*f*1	*p*	*r*	*f*1	*p*	*r*	*f*1
RoBERTa	*k*	.85	.75	.79	.76	.92	.83	.40	.20	.27
	p	.61	.64	.58	.67	.63	.58	.60	.44	.40
	h	1.0	.50	.67	.67	1.0	**.80**	-	-	-
XLNet	*k*	.88	.79	**.80**	.81	.89	**.85**	.60	45	.51
	p	.69	.56	.61	.56	.67	.61	.80	.50	.55
	h	1.0	.50	**.67**	.67	1.0	**.80**	-	-	-
B-Overfl.	*k*	.83	.60	.65	.71	.88	.78	.23	.20	.21
	p	.40	.13	.18	.63	1.0	**.75**	.20	.10	.13
	h	1.0	.50	**.67**	.67	1.0	**.80**	-	-	-
GPT-3	*k*	.92	.63	.73	.78	.95	**.85**	1.0	.67	**.77**
	p	.74	.60	**.67**	.65	.78	.71	.93	1.0	**.96**
	h	1.0	.50	**.67**	.67	1.0	**.80**	-	-	-
Baseline (LR)	*k*	.82	.65	.71	.73	.88	.80	.63	.37	.45
	p	.40	.29	.31	.62	.73	.65	.40	.28	.31
	h	1.0	.50	**.67**	.67	1.0	**.80**	-	-	-
Class Distrib.		*40 + 4*			*68 + 4*			*14 + 0*		

Table 5.23: Evaluation results for the classification of *class-related* child classes

all performance metrics. The difference is most pronounced in the p-fold cross-validation and in on the hold-back projects, where the GPT-3 approach achieved a .26 higher f1-score and .27 higher MCC compared to the second-best approach. It can further be seen that RoBERTa and XLNet still performed moderately in the k-fold cross-validation but produced poor predictions for the hold-back projects and in the p-fold cross-validation. Surprisingly, apart from GPT-3, all transformer models performed worse in p-fold cross-validation compared to the best baseline, suggesting poorer generalisation to unseen projects from these models. The examination of the individual class performances indicates an unreliable classification of the *Tool* class, according to the limited number of samples in this class. All other classes were predicted about equally well, whereas the classification performance of *Framework*, apart from the class tool, was predicted slightly worse in all approaches.

5.4.1.13 Boundary Interface

The deepest local classifier in the taxonomy is the classifier for the design decisions regarding the *Boundary Interfaces*. In comparison, the GPT-3 approach is shown to be the most suitable, as it delivered the best results in all evaluation methods. We find BERTOverflow and XLNet to be the second best, depending on the metric. RoBERTa performed worst among all transformer models for this classifier and did not outperform the baseline in the k-fold cross-validation. Apart from that, all transformer models delivered better results than the baseline. Remarkably, GPT-3 achieved an MCC of .65 when evaluated on the

Technological		*p*	*r*	*f*1	*A*	*MCC*
	k	.74	.75	.74	.85	.81
RoBERTa	*p*	.15	.24	.15	.35	.13
	h	.27	.25	.26	.43	.26
	k	.69	.67	.66	.77	.72
XLNet	*p*	.21	.24	.19	.37	.15
	h	.07	.18	.10	.33	.00
	k	.52	.49	.48	.58	.48
B-Overflow	*p*	.41	.37	.34	.53	.39
	h	.70	.20	.10	.33	.00
	k	.88	.85	**.86**	**.89**	**.87**
GPT-3	*p*	.70	.70	**.69**	**.78**	**.72**
	h	.92	.92	**.92**	**.86**	**.81**
	k	.64	.62	.61	.73	.67
Baseline (SVM)	*p*	.51	.47	.43	.53	.45
	h	.70	.20	.10	.33	.00

Table 5.24: Evaluation results of the *technological* classifier

Technological		**Tool**			**Data Base**			**Platform**		
		p	*r*	*f*1	*p*	*r*	*f*1	*p*	*r*	*f*1
	k	.20	.20	.20	.81	1.0	.89	.91	.81	.85
RoBERTa	*p*	.20	.04	.07	.13	.28	.12	.36	.77	.46
	h	-	-	-	1.0	.25	.40	.40	.37	**1.0**
	k	.20	.20	.20	.77	89	.82	.73	.81	.76
XLNet	*p*	.16	.06	**.08**	.26	.28	.14	.36	.49	.38
	h	-	-	-	.37	.88	.52	.00	.00	.00
	k	.00	.00	.00	.74	.72	.73	.57	.54	.51
B-Overfl.	*p*	.00	.00	.00	.62	.50	.50	.35	.28	.21
	h	-	-	-	.00	.00	.00	.33	1.0	.50
	k	.67	.67	**.67**	.92	1.0	**.95**	.93	.89	**.91**
GPT-3	*p*	.00	.00	.00	.94	.83	**.86**	.73	.85	**.73**
	h	-	-	-	.80	1.0	**.89**	.78	.88	.82
	k	.00	.00	.00	.88	.84	.85	.64	.67	.64
Baseline (SVM)	*p*	.00	.00	.00	.84	.69	.75	.46	.48	.40
	h	-	-	-	.00	.00	.00	.33	1.0	.50
Class Distrib.		*4 + 0*			*43 + 4*			*46 + 8*		

Table 5.25: Evaluation results for the classification of the first three *technological* child classes

Technological		**Boundary**			**Prog. Lang**			**Framework**		
		p	*r*	*f*1	*p*	*r*	*f*1	*p*	*r*	*f*1
RoBERTa	*k*	.87	.86	.86	.91	.85	**.87**	.75	.79	.75
	p	.19	.33	.24	.00	.00	.00	.00	.00	.00
	h	.54	.00	.00	.00	.00	.00	.00	.00	.00
XLNet	*k*	.77	.87	.81	.81	.75	.75	.84	.51	.61
	p	.32	.43	.36	.09	.11	.10	.06	.06	.06
	h	.00	.00	.00	.00	.00	.00	.00	.00	.00
B-Overfl.	*k*	.63	.58	.59	.48	.60	.52	.70	.49	.52
	p	.57	.69	.60	.37	.43	.37	.51	.32	.33
	h	.00	.00	.00	.00	.00	.00	.00	.00	.00
GPT-3	*k*	.82	.94	**.87**	.96	.82	**.87**	.95	.81	**.87**
	p	.69	.71	**.64**	.93	.95	**.93**	.89	.73	**.80**
	h	1.0	.71	**.83**	1.0	1.0	**1.0**	1.0	1.0	**1.0**
Baseline (SVM)	*k*	.69	.82	.74	.88	.64	.72	.72	.72	.72
	p	.59	.75	.60	.62	.54	.44	.55	.36	.42
	h	.00	.00	.00	.00	.00	.00	.00	.00	.00
Class Distrib.		*54 + 7*			*37 + 1*			*34 + 1*		

Table 5.26: Evaluation results for the classification of the last three *technological* child classes

hold-back projects, while all others achieved a value of 0. The data set's distribution of the two classes user *Interface* and *API* is pretty balanced. Nevertheless, the classification performances for the two classes were quite uneven. In k-fold cross-validation, all approaches classified design decisions of the class *API* significantly more reliably than *User Interfaces*, with deltas of up to .44 percentage points in the f1-score. In p-fold cross-validation, BERTOverflow and GPT-3 classify *User Interface* better than *API*. Concerning the evaluation results of the hold-back projects, we see that GPT-3 is the only one that achieves values greater than 0 for the class *User Interface*. In the data set of hold-back projects, there is only one sample of the class *User Interface*. Based on the recall, we see that GPT-3 is the only one that has correctly assigned this sample. However, it has also generated a false positive for this class.

Boundary Int.		p	r	$f1$	A	MCC
RoBERTa	k	.81	.71	.71	.85	.50
	p	.81	.69	.68	.84	.48
	h	.43	.50	.46	.86	.00
XLNet	k	.80	.74	.75	**.87**	.53
	p	.84	.85	**.89**	.84	.75
	h	.43	.50	.46	.86	.00
B-Overflow	k	.70	.68	.67	.82	.68
	p	.73	.60	.60	.90	.80
	h	.43	.50	.46	.86	.00
GPT-3	k	.83	.76	**.79**	.82	**.76**
	p	.78	.69	.73	**.92**	**.86**
	h	.75	.92	.79	.86	**.65**
Baseline (LR)	k	.78	.71	.73	.80	.51
	p	.68	.51	.55	.67	.28
	h	.34	.88	.37	.71	.00

Table 5.27: Evaluation results of the *design decision* classifier

5.4.2 Conclusion for the Local Classifiers

Upon summarising the evaluation of individual local classifiers, we observed that the results differ greatly from one subclass to another and are overall relatively unreliable. In k-fold cross-validation, 3 out of 13 classifiers achieved an MCC greater than .80, and 5 out of 13 achieved an accuracy greater than .90. In p-fold cross-validation, only one classifier achieved an MCC greater than .80, and the same one was the only one to achieve an accuracy greater than .90. In general, we found that p-fold cross-validation yields worse results compared to k-fold cross-validation for all classifiers.

An observation that applies to almost all classifiers is that the performance of the individual classes correlates with the size of class distributions. This is evident, for example,

Boundary Int.		**User Interface**			**API**		
		p	*r*	*f*1	*p*	*r*	*f*1
RoBERTa	*k*	.76	.45	.52	.85	.97	.90
	p	.80	.37	.47	.82	1.0	.90
	h	.00	.00	.00	.86	1.0	**.92**
XLNet	*k*	.75	.50	.59	.85	.97	**.91**
	p	.95	.70	.77	.89	.98	**.93**
	h	.00	.00	.00	.86	1.0	**.92**
B-Overfl.	*k*	.60	.37	.45	.80	1.0	.89
	p	.92	.73	.73	.54	.47	.47
	h	.00	.00	.00	.86	1.0	**.92**
GPT-3	*k*	.83	.58	**.67**	.83	.94	.88
	p	1.0	.72	**.82**	.55	.67	.60
	h	.50	1.0	**.67**	1.0	.83	.91
Baseline (LR)	*k*	.80	.50	.59	.82	.93	.86
	p	.89	.45	.59	.47	.57	.51
	h	.00	.00	.00	.67	.83	.74
Class Distrib.			*16 + 1*			*38 + 6*	

Table 5.28: Evaluation results for the classification of *boundary interface* child classes

in the local classifiers for the *Design Decision* classes, where *Property Decisions* are classified significantly worse than *Existence Decisions* and *Executive Decisions.*

Another finding that is shared by almost all local classifiers is that the classifiers show inferior generalisation when evaluated with the p-fold cross-validation. A clear example is the classification of *Framework* by the *Technological* classifier, where a significant discrepancy between k-fold and p-fold is seen across all classifiers except GPT-3. This outcome was anticipated, as there may be correlations or patterns in design decisions that occur within a project but may not be present in other projects. For example, if the JavaScript framework *"Express"* occurs in one project but not in other projects, then this framework will eventually appear in the test data set but not in the training data set. If the transformer model does not already know *"Express"* as a framework from the outset, it has no chance of learning this classification. With a view to the evaluation of the hold-back project data, we observe strongly fluctuating results.

The results sometimes lay between those of p-fold and k-fold but sometimes clearly above or below them. First of all, it should be noted that the hold-back projects are also unseen projects, which means that some of the same effects can be expected as with the p-fold cross-validation. The highly fluctuating results are likely attributed to the fact that the distribution of classes in the test data set can be vastly different from the distribution in the training data set, unlike stratified cross-validation. Additionally, there are no multiple testing runs with averaged results that reduce the variance, in contrast to cross-validations. Therefore, the evaluation on hold-back projects is less consistent for performance prediction but rather highlights the issue of highly fluctuating and unreliable classifications.

When comparing the performances of the transformers to those of the baseline, we find that the transformer approaches generally perform better. However, BERTOverflow performs significantly worse than XLNet, GPT-3, and RoBERTa in most cases. During k-fold cross-validation, BERTOverflow often provides only slightly better results than the best baseline but generalises better to unseen projects. On the *Class-Related* and *Technological* levels of local classifiers, BERTOverflow performs worse than the baseline.

If we compare XLNet, RoBERTa, and GPT-3 with each other, it is difficult to determine a clear winner. There is no language model that visibly dominates. In many cases, the best model depends on which metrics are given the most weight and which validation method we focus on. However, if we evaluate the performance solely based on the MCC, GPT-3 is most often the best model, as shown in Table 5.29.

Classifier Level	Method	Best Model	Δ to 2nd
Root	*k*-fold	GPT-3, XLNet	.07
	p-fold	GPT-3	.05
	hold-back	RoBERTa	.08
Design Dec.	*k*-fold	XLNet	.03
	p-fold	RoBERTa	.09
	hold-back	RoBERTa	.02
Existence Dec.	*k*-fold	RoBERTa	.03
	p-fold	GPT-3	.01
	hold-back	XLNet	.04
Property Dec.	*k*-fold	Baseline (NB)	.06
	p-fold	BERTOverflow	.05
	hold-back	all .00	-
Executive Dec.	*k*-fold	all .00	-
	p-fold	all .00	-
	hold-back	all .00	-
Structural Dec.	*k*-fold	RoBERTa	.01
	p-fold	RoBERTa, GPT-3	.04
	hold-back	GPT-3	.40
Arrangement Dec.	*k*-fold	BERTOverflow	.18
	p-fold	BERTOverflow	.25
	hold-back	XLNet	.37
Behavioural Dec.	*k*-fold	RoBERTa	.02
	p-fold	GPT-3	.12
	hold-back	GPT-3, XLNet	.07
Extra-Systemic	*k*-fold	GPT-3	.03
	p-fold	GPT-3, XLNet	.55
	hold-back	XLNet	.59
Intra-Systemic	*k*-fold	GPT-3	.02
	p-fold	BERTOverflow	.14
	hold-back	GPT-3	.48
Class-Related	*k*-fold	XLNet	.03
	p-fold	GPT-3	.09
	hold-back	all identical	-
Technological	*k*-fold	GPT-3	.06
	p-fold	GPT-3	.27
	hold-back	GPT-3	.42
Boundary Interf.	*k*-fold	GPT-3	.08
	p-fold	GPT-3	.06
	hold-back	GPT-3	.65

Table 5.29: Best performing models for each evaluation method according to MCC

5.4.3 Global Classifier

So far, we have examined the validation results of each local classifier. The local classifiers were developed with the goal of assembling a global hierarchical LCPN classifier that classifies text phrases into one of 25 leaf node classes. In the following, we analyse the evaluation results of the global flat classifiers and the global LCPN classifier to investigate whether the hierarchical approach brings an advantage.

5.4.3.1 Baselines

First, we analyse the behaviour of the global flat and hierarchical classifiers with the baseline approaches. An analysis of the cross-validation results reveals no significant performance disparities between the LCPN and flat approaches with the logistic regression and SVM methods. The only noteworthy difference is that the logistic regression with the hierarchical classifier produced a higher hierarchical precision.

For the naive Bayes approach, we observe only a slight improvement in performance with the hierarchical approach compared to the flat approach. However, a clear superiority of the hierarchical classifier is observed with the random forest baseline, where the hierarchical classifier achieved a .13 percentage point higher MCC and .6 percentage point higher accuracy in the k-fold cross-validation. The improvements in the hierarchical recall are also noteworthy, indicating that the predictions deviate less from the actual classifications overall.

Looking at the validation results on unseen projects, we observe that the random forest showed noticeably better performance with the LCPN classifier. Further, we observed improvements in the logistic regression concerning hierarchical metrics on unseen projects. Both SVM and logistic regression performed significantly better on the hold-back data with the hierarchical classifier than the Flat ones but still remained at a very low performance on these data.

Overall, we find that in the case of baselines, the global hierarchical classifiers are generally slightly superior to the global flat classifiers. Among all baseline classifiers, the LCPN classifiers with naive Bayes and SVM performed best in the validation on unseen projects. In terms of macro-averaged f1-score, accuracy, and MCC, the flat classifier of logistic regression was strongest in k-fold cross-validation. However, when taking into account the hierarchical metrics as well, we deem the hierarchical classifiers of random forest, SVM, and logistic regression superior.

5.4.3.2 Transformer Models

Having concluded a tendency towards an advantage of hierarchical classifiers among the baseline approaches, we now turn to the global classifiers based on pre-trained transformer LMs.

To start with, we review the results of the k-fold cross-validation. We observe that XLNet showed no substantial performance differences between the hierarchical and flat classifiers. However, XLNet performed with a slightly higher MCC and accuracy with the LCPN classifier. For RoBERTa, BERTOverflow and GPT-3, we observe more pronounced differences in the k-fold cross-validation. With all three transformer models, the flat

Log. Regr.		p	r	$f1$	p_H	r_H	$f1_H$	A	MCC
	k	.27	.24	.25	.56	**.60**	**.59**	.50	.38
Hierarchical	p	.13	**.11**	.12	**.57**	**.54**	**.56**	.39	.21
	h	**.14**	**.22**	**.17**	**.29**	.27	**.28**	**.22**	**.09**
	k	**.31**	.24	**.27**	.54	.44	.48	**.53**	**.39**
Flat	p	**.14**	**.11**	**.13**	.56	.39	.46	**.41**	**.22**
	h	.01	.07	.03	.14	.12	.13	.14	.00

Table 5.30: Evaluation results of the global classifiers of Log. Regres.

Naive Bayse		p	r	$f1$	p_H	r_H	$f1_H$	A	MCC
	k	**.27**	**.29**	**.28**	.55	**.60**	**.57**	**.46**	**.36**
Hierarchical	p	**.18**	**.20**	**.19**	.51	**.51**	**.51**	**.40**	**.27**
	h	**.08**	.04	**.05**	**.21**	**.19**	**.20**	**.16**	.00
	k	.25	.25	.25	**.57**	**.60**	**.57**	.43	.31
Flat	p	.16	.17	.17	**.52**	**.51**	**.51**	.37	.21
	h	.01	**.06**	.02	.16	.11	.10	.13	.00

Table 5.31: Evaluation results of the global classifiers of Naive Bayes.

SVM		p	r	$f1$	p_H	r_H	$f1_H$	A	MCC
	k	**.25**	**.24**	**.25**	.55	**.63**	**.59**	.48	**.38**
Hierarchical	p	**.15**	**.13**	**.14**	.47	.45	.46	.39	.23
	h	**.11**	**.05**	**.07**	**.24**	**.32**	**.27**	**.24**	**.11**
	k	**.25**	.23	.24	**.56**	.61	.58	**.49**	**.38**
Flat	p	.12	.11	.11	**.50**	**.50**	**.50**	**.40**	**.24**
	h	.08	**.05**	.02	.21	.11	.17	.19	.02

Table 5.32: Evaluation results of the global classifiers of SVM.

Rand. Forest		p	r	$f1$	p_H	r_H	$f1_H$	A	MCC
	k	.25	**.23**	**.24**	.56	**.61**	**.58**	**.49**	**.38**
Hierarchical	p	.12	**.11**	**.11**	**.50**	**.50**	**.50**	**.40**	**.24**
	h	**.11**	.05	.07	.24	**.32**	**.27**	**.24**	**.10**
	k	**.27**	.20	.23	.55	.49	.52	.43	.25
Flat	p	**.13**	.09	**.11**	**.50**	.40	.44	.38	.16
	h	.01	**.06**	.02	.16	.11	.10	.13	.00

Table 5.33: Evaluation results of the global classifiers of Random Forest.

approach performed better on k-fold cross-validation, outperforming the hierarchical approach in all metrics.

Examining the validation results for the p-fold cross-validation, the expectation that the p-fold cross-validation would generally produce worse results than the k-fold cross-validation is confirmed. However, it appears that the LCPN classifiers could considerably improve performance for all transformers, except for GPT-3. For instance, XLNet improved its MCC from .16 to .35 and its macro-averaged f1-score from .08 to .14, using the LCPN classifier compared to the flat classifier. When validating on the Hold-Back Data, an even more drastic improvement was observed with RoBERTa, XLNet, and BERTOverflow. Here, the MCC for BERTOverflow improved from .00 to .48. The performance of XLNet and RoBERTa also improved significantly. On the contrary, this behaviour was not observed with GPT-3. The LCPN classifier performed no better than the flat classifier in the hold-back data with GPT-3. In the p-fold cross-validation, the flat classifier showed to be visibly better.

In conclusion, after comparing all the global classifiers, the flat RoBERTa approach produced the best results in k-fold cross-validation. In p-fold cross-validation, GPT-3 with the flat approach yielded the best results. Lastly, in validation on hold-back data, the hierarchical BERTOverflow and flat GPT-3 approach performed best, with GPT-3 providing the higher macro-averaged f1-score, while BERTOverflow achieved the higher hierarchical f1-score.

RoBERTa		p	r	$f1$	p_H	r_H	$f1_H$	A	MCC
	k	.43	.44	.44	.63	.76	.69	.59	.54
Hierarchical	p	**.19**	**.21**	**.20**	.58	**.68**	**.63**	**.46**	**.37**
	h	**.26**	**.26**	**.26**	$\underline{.73}$	$\underline{\mathbf{.73}}$	$\underline{\mathbf{.73}}$	**.54**	**.48**
	k	$\underline{\mathbf{.54}}$	$\underline{\mathbf{.53}}$	$\underline{\mathbf{.53}}$	$\underline{\mathbf{.79}}$	$\underline{\mathbf{.81}}$	$\underline{\mathbf{.80}}$	$\underline{\mathbf{.71}}$	$\underline{\mathbf{.64}}$
Flat	p	.10	.12	.10	**.62**	.53	.57	.44	.26
	h	.04	.06	.05	.51	.51	.51	.18	.12

Table 5.34: Evaluation results of the global classifiers of RoBERTa.

XLNet		p	r	$f1$	p_H	r_H	$f1_H$	A	MCC
	k	**.46**	.42	**.44**	**.75**	**.78**	.76	**.70**	**.63**
Hierarchical	p	**.15**	**.13**	**.14**	.59	**.58**	**.59**	**.55**	**.35**
	h	**.26**	**.24**	**.25**	**.67**	**.68**	$\underline{\mathbf{.68}}$	**.39**	**.35**
	k	**.46**	**.44**	**.44**	**.75**	**.78**	**.77**	.68	.61
Flat	p	.01	.10	.08	**.61**	.52	.55	.43	.16
	h	.02	.05	.02	.51	.46	.48	.20	.12

Table 5.35: Evaluation results of the global classifiers of XLNet.

B-Overflow		p	r	$f1$	p_{H}	r_{H}	$f1_{\mathrm{H}}$	A	MCC
Hierarchical	k	.33	.34	.34	.53	.66	.59	.49	.34
	p	**.19**	**.21**	**.20**	.58	**.58**	.53	**.36**	**.27**
	h	**.26**	**.26**	**.26**	**.73**	**.73**	**.73**	**.54**	**.48**
Flat	k	**.43**	**.44**	**.40**	.75	**.77**	.75	**.59**	**.54**
	p	.09	.10	.08	**.61**	.52	**.55**	.43	.16
	h	.01	.05	.02	.20	.07	.10	.20	.00

Table 5.36: Evaluation results of the global classifiers of BERTOverflow.

GPT-3		p	r	$f1$	p_{H}	r_{H}	$f1_{\mathrm{H}}$	A	MCC
Hierarchical	k	.41	.44	.42	.61	.64	.62	.63	.59
	p	.33	.31	.32	.62	.59	.60	.60	.44
	h	.34	.40	.37	.64	**.70**	**.67**	**.55**	**.49**
Flat	k	**.52**	**.46**	**.49**	**.76**	**.75**	**.76**	**.70**	**.63**
	p	**.34**	**.32**	**.33**	**.73**	**.70**	**.71**	**.65**	**.56**
	h	**.37**	**.43**	**.40**	**.67**	.67	**.67**	.54	.48

Table 5.37: Evaluation results of the global classifiers of GPT-3.

6 Conclusion

In this thesis, we developed and evaluated various methods for automatically classifying design decisions. We developed hierarchical classifiers using the LCPN strategy and compared them with flat classifiers that ignore the class hierarchy. Furthermore, we applied the pre-trained language models RoBERTa, XLNet, BERTOverflow and GPT-3 for both the flat and hierarchical approaches. In the evaluation, we assessed the classification performance of the pre-trained transformer language model-based approaches and baseline approaches. To do so, we first examined each local classifier of the LCPN approach. Then, we analysed and compared the results of both the hierarchical global classifiers and the flat global classifiers. We will now summarise these results by addressing the research questions.

RQ1: *Are there parent classes or hierarchy levels in the taxonomy at which classification performance is proportionally high or low?*

The answer to this question depends on how we weigh the performance metrics. If we focus on the macro-averaged f1-Score and the MCC, the classifiers with a high class imbalance in the data set perform poorly, especially the *Executive Decision* and *Design Decision* classifiers. For instance, in the case of the *Design Decision*, although there is a considerable amount of training data available, the child class *Property* only makes up 7% of the three child classes, resulting in a decrease in the classification performance in terms of MCC and f1-score. However, this property does not have a high effect to the accuracy since the poorly predicted classes only make up a small proportion of all training samples.

A similar problem occurs with the local classifiers of the *Property Decision* and *Arrangement Decision.* Here, the class imbalance is not as pronounced, but there are classes with only 10 and 4 training samples, respectively, leading to a poor classification of the underrepresented classes and a decrease of the overall performance in terms of MCC and macro-averaged f1.

If we give more weight to accuracy as a performance metric, the *Technological* class stands out as particularly difficult, performing poorly, particularly on unseen projects. Here, all approaches, except for GPT-3, showed a MCC of less than .50 and an accuracy of at most .53 on unseen projects. The sub-data set of the local classifier *Technological* includes, apart from the subclass tool, more than 30 training samples for all classes and relatively equal distribution of classes. Therefore, the results of this local classifier suggest that the difficulty in classifying this class is not solely attributed to the data set but is also inherent to the classes themselves.

RQ2 : *What is the effect of a hierarchical classification approach versus a flat approach on the overall classification result, with conventional methods and with the transformer approaches?*

After discussing the evaluation results of all local classifiers, we analysed and compared the hierarchical and flat global classifiers, allowing us to address RQ2. In the baseline approaches, the results between hierarchical and flat classifiers were often close to each other, with the hierarchical approaches mostly outperforming the flat ones slightly. Considering all metrics, we hence consider the hierarchical approaches superior to the baselines.

For the Transformer models, we observe some larger differences between LCPN and flat classifiers. In k-fold cross-validation, there was no significant difference for XLNet, but for other Transformer models, the flat approach performed noticeably better. In validation using p-Fold cross-validation and hold-back data, XLNet, BERTOverflow, and RoBERTa significantly improved with the hierarchical approach. However, GPT-3 did not show this behaviour and performed equally well on unseen projects with both the flat and hierarchical approach, with a slight advantage for the flat approach.

In summary, we cannot make a clear statement as to whether the hierarchical approach is superior to the flat approach, as this varies both from model to model and from validation method to validation method. The evaluation results suggest, however, that the hierarchical approach improves performance on unseen projects for RoBERTa, XLNet, and BERTOverflow, indicating better generalisation.

RQ3 : *How and to what extend do pre-trained LMs improve classification performance over conventional approaches?* The evaluation of the local classifiers showed that the pre-trained transformer language models almost invariably performed better than the baseline. This result was confirmed when assessing the hierarchical global models, where the transformer-based models performed better than the baselines overall. The advantage was particularly noticeable during the validation on the hold-back data.

The transformer-based approaches also showed significantly better performance during k-fold cross-validation for the flat global models. However, during p-fold cross-validation, the flat classifiers of XLNet and BERTOverflow performed worse than most baseline approaches. GPT-3 stands out, delivering significantly better results with all validation methods compared to the baseline approaches.

Considering all these results, there were cases where the baseline approaches performed better than some transformer-based approaches. Ultimately, however, we still find the transformer-based models as leading due to their better performance in most of the local classifiers and the resulting hierarchical classifiers. Further, in particular, GPT-3 delivered the best global results, outperforming every baseline with both the hierarchical and the flat approach.

RQ4 : *Which of the applied LMs leads to the best classification results?* Finally, we address RQ4. Upon evaluation of the results of local classifiers, we found that BERTOverflow performed worst compared to other transformer models. Further, we observed no clear winner among XLNet, RoBERTa, and GPT-3. However, when evaluated based on MCC scores, GPT-3 was found to be the most frequent leader.

When looking at the evaluation results of hierarchical classifiers, these findings are supported. The transformer models produced similar good results, with the hierarchical model of GPT-3 performing slightly better on unseen projects, while the hierarchical model of RoBERTa performed best in k-fold cross-validation. The hierarchical classifier with BERTOverflow performed lower than the other transformers.

When including the global flat classifiers, RoBERTa was found to have the best result in k-fold cross-validation, with GPT-3 slightly behind. However, GPT-3 was superior in p-fold cross-validation and on the hold-back data.

Out of all global models, GPT-3 produced the second-best result in k-fold cross-validation with only a slight deficit to RoBERTa, and the clearly best results during validation on unseen projects. Therefore, we see GPT-3 pre-trained transformer language model for classifying design decisions.

6.1 Threads to Validity

In this work, we identify several factors that threaten the validity of our experiments and research. In this section, we discuss these potential threats to validity.

Data Set An essential factor that jeopardises the validity of our work attributes to our data sets. Although the data sets include projects from different domains, it is unclear how representative the software architecture documentations of these projects are and whether the data set may be subject to certain biases. Further, all projects in our data set are open-source. The documentations of commercial closed-source projects are may be written and constructed differently, which could decrease the transferability of our results to other projects. An indicator of this issue is that the validation on our hold-back data set sometimes yielded results that deviated clearly from those obtained by cross-validation methods.

Another penalty of our data sets pertain to their size and class imbalances. Especially with local classifiers using a particularly small sub-data set with underrepresented classes, it remains unclear to what extent the measured performance can be attributed to these characteristics of the data set. To address these problems, the data sets should be expanded to ensure at least a minimum number of samples per class.

Labelling The data set was manually labelled by one student. Despite the conscientious execution of the labelling process, inconsistent or erroneous labels may occur. In particular, if underrepresented classes in the data set were inconsistently or incorrectly labelled, this could have a pronounced negative impact on the classifiers' performance. This risk could be mitigated by a more extensive labelling process involving multiple labellers or quality control measures.

Hyperparameters The training hyperparameters can significantly influence the performance of a neural network. Therefore, we conducted a separate hyperparameter search for each transformer-based classifier, except for GPT-3, using a Bayesian optimiser. However, hyperparameter search only provides an approximation of the optimal hyperparameters.

Thus, we do not know if and to what extent the potential of each transformer language model was utilised. Thus, there is a chance that performance differences between the different Transformer LMs are due to their hyperparameters rather than their capabilities. Specifically for GPT-3, there is a risk that the hyperparameters were not optimal since we waived a hyperparameter search.

Overfitting Another risk threat to validity is overfitting. In case of overfitting, the model is too closely adapted to the training data and therefore generalises much worse. Although we have implemented a form of early stopping as a countermeasure, the risk of overfitting remains. This pitfall is related to that of the training hyperparameters since they can be one of the causes. A larger data set would be a step towards reducing the risk of overfitting.

Cross Validation For the p-fold and the k-fold cross-validation we applied in our evaluation, we use a five-way split in each case, i.e. k=5 and p=5. We made this choice primarily for computing time reasons. Since the number of folds influences the performance estimation, the performance estimation might not be optimal in our case. A higher number of folds, e.g. 10-fold, may improve the performance estimation since more training data is available in the individual test runs.

In addition, to split the data into individual folds, we also use a fixed random seed to maintain the reproducibility of the results. However, the choice of the seed may also has an impact on the results, which we have taken into account. One way to improve the validation process in this regard would be to perform the cross-validation multiple times with different random seeds and average the results.

6.2 Future Work

The results of this study can serve as a starting point for further research in this direction. As stated in conclusion and the threads to validity, the results of this study are strongly limited by the size and imbalance of the data set. Therefore, the most crucial future work is to conduct evaluations on significantly larger data sets to provide more reliable answers to our research questions and to increase the validity of the results.

The measured performances of our approach are currently deficient. Although these results are partly due to a scarce data set, we see the potential to improve our approach. The application of transfer learning using pre-trained language models has been shown to be advantageous. However, we could not find a clear advantage of the LCPN strategy in combination with transfer learning. Therefore, we propose combining pre-trained transformer language models with alternative classification strategies, such as binary classification, for future work.

In this thesis, we only addressed multi-class classification, assigning each line of text in the documentation to exactly one class. However, these lines sometimes contain multiple design decisions. In order to fully benefit from the automatic classification of design decisions in software architecture documentation, a multi-label classification should be pursued. Therefore, a desirable future work is to extend the approach to a multi-label setting.

Bibliography

[1] Leonidas Akritidis and Panayiotis Bozanis. “How Dimensionality Reduction Affects Sentiment Analysis NLP Tasks: An Experimental Study”. In: *Artificial Intelligence Applications and Innovations: 18th IFIP WG 12.5 International Conference, AIAI 2022, Hersonissos, Crete, Greece, June 17–20, 2022, Proceedings, Part II.* Springer. 2022, pp. 301–312.

[2] Ali Araabi, Christof Monz, and Vlad Niculae. “How Effective is Byte Pair Encoding for Out-Of-Vocabulary Words in Neural Machine Translation?” In: *arXiv preprint arXiv:2208.05225* (2022).

[3] Adailton Araujo et al. “From bag-of-words to pre-trained neural language models: Improving automatic classification of app reviews for requirements engineering”. In: *Anais do XVII Encontro Nacional de Inteligência Artificial e Computacional.* SBC. 2020, pp. 378–389.

[4] Adailton Ferreira de Araújo and Ricardo Marcondes Marcacini. “Re-bert: automatic extraction of software requirements from app reviews using Bert language model”. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing.* 2021, pp. 1321–1327.

[5] Victor R Basili and David M Weiss. “A methodology for collecting valid software engineering data”. In: *IEEE Transactions on software engineering* 6 (1984), pp. 728–738.

[6] Himanshu Batra et al. “BERT-Based Sentiment Analysis: A Software Engineering Perspective”. In: *International Conference on Database and Expert Systems Applications.* Springer. 2021, pp. 138–148.

[7] Daniel Berrar. “Bayes’ theorem and naive Bayes classifier”. In: *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics* 403 (2018), p. 412.

[8] Manoj Bhat et al. “Automatic extraction of design decisions from issue management systems: a machine learning based approach”. In: *European Conference on Software Architecture.* Springer. 2017, pp. 138–154.

[9] Eeshita Biswas et al. “Achieving reliable sentiment analysis in the software engineering domain using bert”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2020, pp. 162–173.

[10] Alexander Brinkmann and Christian Bizer. “Improving Hierarchical Product Classification using Domain-specific Language Modelling.” In: *IEEE Data Eng. Bull.* 44.2 (2021), pp. 14–25.

[11] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: (2020). arXiv: `2005.14165 [cs.CL]`.

[12] Jamie Callan et al. *Clueweb09 data set.* 2009.

[13] Michelle Y Chew et al. "A comparative study of name entity recognition techniques in software engineering texts". In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. 2022, pp. 1611–1614.

[14] *Common Crawl.* URL: `https://commoncrawl.org/`.

[15] Zihang Dai et al. "Transformer-xl: Attentive language models beyond a fixed-length context". In: *arXiv preprint arXiv:1901.02860* (2019).

[16] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[17] Sheng Ding. "Feature selection based F-score and ACO algorithm in support vector machine". In: *2009 Second International Symposium on Knowledge Acquisition and Modeling*. Vol. 1. IEEE. 2009, pp. 19–23.

[18] Susan Dumais and Hao Chen. "Hierarchical classification of web content". In: *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval.* 2000, pp. 256–263.

[19] *Fine-tuning - OpenAI API.* URL: `https://platform.openai.com/docs/guides/fine-tuning`.

[20] Liming Fu et al. "A Machine Learning Based Ensemble Method for Automatic Multiclass Classification of Decisions". In: *Evaluation and Assessment in Software Engineering*. 2021, pp. 40–49.

[21] Susan Gauch, Aravind Chandramouli, and Shankar Ranganathan. "Training a hierarchical classifier using inter document relationships". In: *Journal of the American Society for Information Science and Technology* 60.1 (2009), pp. 47–58.

[22] Aaron Gokaslan and Vanya Cohen. *Openwebtext corpus.* 2019. URL: `http://web.archive.org/save/http://Skylion007.github.io/OpenWebTextCorpus` (visited on 01/26/2023).

[23] Jan Gorodkin. "Comparing two K-category assignments by a K-category correlation coefficient". In: *Computational biology and chemistry* 28.5-6 (2004), pp. 367–374.

[24] Margherita Grandini, Enrico Bagli, and Giorgio Visani. "Metrics for multi-class classification: an overview". In: *arXiv preprint arXiv:2008.05756* (2020).

[25] Tobias Hey et al. "NoRBERT: Transfer learning for requirements classification". In: *2020 IEEE 28th International Requirements Engineering Conference (RE).* IEEE. 2020, pp. 169–179.

[26] Lazaros Iliadis and Ilias Maglogiannis. *Artificial Intelligence Applications and Innovations: 12th IFIP WG 12.5 International Conference and Workshops, AIAI 2016, Thessaloniki, Greece, September 16-18, 2016, Proceedings.* Vol. 475. Springer, 2016.

[27] Maliheh Izadi. "CatIss: An Intelligent Tool for Categorizing Issues Reports using Transformers". In: *arXiv preprint arXiv:2203.17196* (2022).

[28] Anton Jansen and Jan Bosch. “Software architecture as a set of architectural design decisions”. In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA’05)*. IEEE. 2005, pp. 109–120.

[29] Alyssa Josephs, Fabian Gilson, and Matthias Galster. “Towards Automatic Classification of Design Decisions from Developer Conversations”. In: *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2022, pp. 10–14.

[30] Daniel Jurafsky and James H Martin. “N-gram language models”. In: *Speech and language processing* 23 (2018).

[31] Daniel Jurafsky and James H Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* 2023.

[32] Jan Keim and Anne Koziolek. “Towards consistency checking between software architecture and informal documentation”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2019, pp. 250–253.

[33] Jan Keim et al. “A Taxonomy for Design Decisions in Software Architecture Documentation”. In: (2022).

[34] Salman Khan et al. “Transformers in vision: A survey”. In: *ACM computing surveys (CSUR)* 54.10s (2022), pp. 1–41.

[35] Ron Kohavi et al. “A study of cross-validation and bootstrap for accuracy estimation and model selection”. In: *Ijcai*. Vol. 14. 2. Montreal, Canada. 1995, pp. 1137–1145.

[36] Daphne Koller and Mehran Sahami. *Hierarchically classifying documents using very few words.* Tech. rep. Stanford InfoLab, 1997.

[37] Philippe Kruchten. “An ontology of architectural design decisions in software intensive systems”. In: *2nd Groningen workshop on software variability*. Citeseer. 2004, pp. 54–61.

[38] Taku Kudo and John Richardson. “Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing”. In: *arXiv preprint arXiv:1808.06226* (2018).

[39] Guokun Lai et al. “Race: Large-scale reading comprehension dataset from examinations”. In: *arXiv preprint arXiv:1704.04683* (2017).

[40] Zhenzhong Lan et al. “Albert: A lite bert for self-supervised learning of language representations”. In: *arXiv preprint arXiv:1909.11942* (2019).

[41] Xueying Li, Peng Liang, and Zengyang Li. “Automatic identification of decisions from the hibernate developer mailing list”. In: *Proceedings of the Evaluation and Assessment in Software Engineering*. 2020, pp. 51–60.

[42] Yinhan Liu et al. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[43] Bonan Min et al. “Recent advances in natural language processing via large pretrained language models: A survey”. In: *arXiv preprint arXiv:2111.01243* (2021).

[44] Sebastian Nagel. *Cc-news*. 2016. URL: http://web.archive.org/save/http://commoncrawl.org/2016/10/newsdataset-available (visited on 01/26/2023).

[45] William S Noble. "What is a support vector machine?" In: *Nature biotechnology* 24.12 (2006), pp. 1565–1567.

[46] Robert Parker et al. *English gigaword fifth edition ldc2011t07 (tech. rep.)* Tech. rep. Technical Report. Linguistic Data Consortium, Philadelphia, 2011.

[47] David Lorge Parnas. "Precise documentation: The key to better software". In: *The Future of Software Engineering*. Springer, 2011, pp. 125–148.

[48] Xipeng Qiu et al. "Pre-trained models for natural language processing: A survey". In: *Science China Technological Sciences* 63.10 (2020), pp. 1872–1897.

[49] Alec Radford et al. "Improving language understanding by generative pre-training". In: (2018).

[50] Pranav Rajpurkar et al. "Squad: 100,000+ questions for machine comprehension of text". In: *arXiv preprint arXiv:1606.05250* (2016).

[51] Abhishek Sainani et al. "Extracting and classifying requirements from software engineering contracts". In: *2020 IEEE 28th International Requirements Engineering Conference (RE)*. IEEE. 2020, pp. 147–157.

[52] *scikit-learn: machine learning in Python — scikit-learn 1.2.1 documentation*. URL: https://scikit-learn.org/stable/.

[53] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural machine translation of rare words with subword units". In: *arXiv preprint arXiv:1508.07909* (2015).

[54] Thomas R. Shultz et al. "Curse of Dimensionality". In: *Encyclopedia of Machine Learning*. Springer US, 2011, pp. 257–258. DOI: 10.1007/978-0-387-30164-8_192. URL: https://doi.org/10.1007/978-0-387-30164-8_192.

[55] Carlos N Silla and Alex A Freitas. "A survey of hierarchical classification across different application domains". In: *Data Mining and Knowledge Discovery* 22.1 (2011), pp. 31–72.

[56] Roger Alan Stein, Patricia A Jaques, and Joao Francisco Valiati. "An analysis of hierarchical text classification using word embeddings". In: *Information Sciences* 471 (2019), pp. 216–232.

[57] Xiangchenyang Su and Fang Liu. "A survey for study of feature selection based on mutual information". In: *2018 9th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*. IEEE. 2018, pp. 1–4.

[58] Jeniya Tabassum et al. "Code and named entity recognition in stackoverflow". In: *arXiv preprint arXiv:2005.01634* (2020).

[59] Trieu H Trinh and Quoc V Le. "A simple method for commonsense reasoning". In: *arXiv preprint arXiv:1806.02847* (2018).

[60] Jakob Uszkoreit. *Transformer: A Novel Neural Network Architecture for Language Understanding*. 2017. URL: https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html (visited on 01/16/2023).

[61] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[62] Prateek Verma and Jonathan Berger. “Audio transformers: Transformer architectures for large scale audio understanding. adieu convolutions”. In: *arXiv preprint arXiv:2105.00335* (2021).

[63] Alex Wang et al. “GLUE: A multi-task benchmark and analysis platform for natural language understanding”. In: *arXiv preprint arXiv:1804.07461* (2018).

[64] Andreas S Weigend, Erik D Wiener, and Jan O Pedersen. “Exploiting hierarchy in text categorization”. In: *Information Retrieval* 1.3 (1999), pp. 193–216.

[65] Thomas Wolf et al. “Huggingface’s transformers: State-of-the-art natural language processing”. In: *arXiv preprint arXiv:1910.03771* (2019).

[66] Junfang Wu, Chunyang Ye, and Hui Zhou. “BERT for sentiment classification in software engineering”. In: *2021 International Conference on Service Science (ICSS)*. IEEE. 2021, pp. 115–121.

[67] Yonghui Wu et al. “Google’s neural machine translation system: Bridging the gap between human and machine translation”. In: *arXiv preprint arXiv:1609.08144* (2016).

[68] Zhilin Yang et al. “Xlnet: Generalized autoregressive pretraining for language understanding”. In: *Advances in neural information processing systems* 32 (2019).

[69] Yujia Zhai et al. “A chi-square statistics based feature selection method in text classification”. In: *2018 IEEE 9th International conference on software engineering and service science (ICSESS)*. IEEE. 2018, pp. 160–163.

[70] Cha Zhang and Yunqian Ma. *Ensemble machine learning: methods and applications*. Springer, 2012.

[71] Ting Zhang et al. “Sentiment analysis for software engineering: How far can pre-trained transformer models go?” In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2020, pp. 70–80.

[72] Xiang Zhang, Junbo Zhao, and Yann LeCun. “Character-level convolutional networks for text classification”. In: *Advances in neural information processing systems* 28 (2015).

[73] Yukun Zhu et al. “Aligning books and movies: Towards story-like visual explanations by watching movies and reading books”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 19–27.

[74] Eric R Ziegel. *The elements of statistical learning*. 2003.