Karlsruher Institut für Technologie

Bachelor thesis

Active Partial Label Learning for Parallel Benchmarking

Moritz Brandt

Date: 23. Oktober 2024

Supervisors: Prof. Dr. Peter Sanders
Dr.rer.nat. Markus Iser
M.Sc. Tobias Fuchs

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Benchmarking is an essential part of algorithm development. This also applies to solvers for the SAT (propositional satisfiability) problem. Comparing a new solver to an established portfolio of solvers often requires benchmarking it on large amounts of difficult instances. Dynamic benchmark selection strategies utilizing AL (active learning) can reduce the number of benchmark instances required to accurately estimate the rank of a new solver. However, these strategies do not run benchmark experiments in parallel, which limits their real world applicability. In this thesis, we present a parallelizable AL strategy for dynamic benchmark selection. We evaluate our approach on the Anniversary Track dataset from the 2022 SAT Competition. Our approach can determine a new solver's PAR-2 ranking with about 91% accuracy while only needing 13.5% of the CPU-runtime of a full evaluation and running up to 64 benchmark experiments in parallel.

Acknowledgments

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Contents

1 Introduction

Benchmarking is essential for evaluating, optimizing and comparing algorithms. Especially for $\mathcal{NP}$-complete problems like propositional satisfiability (SAT) that do not have known efficient solutions. Here, benchmarks are the best measure of an algorithm's performance[5]. Since all known SAT solvers have exponential worst-case complexity, benchmarking SAT solvers can be very runtime intensive. While this may be feasible for benchmarking singular algorithm instantiations, algorithm engineering often requires extensive experimentation with different approaches and parameter choices. Therefore, benchmark runs can quickly become a bottleneck for SAT solver development. Definition 1 describes the *New-Solver Problem*, which is often encountered in this context.

Definition 1 (New-Solver Problem)**.** *Given solvers $\mathcal{A}$, instances $\mathcal{I}$, runtimes $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$ with time-limit τ, and a new solver $\hat{a} \notin \mathcal{A}$, incrementally select benchmark instances from $\mathcal{I}$ to maximize the confidence in predicting the rank of $\hat{a}$ while minimizing the total benchmark runtime. [6]*

Problem 1 can be approached with Active Learning (AL). AL is a machine learning technique employed in environments where acquiring labeled training data is expensive. Instead of relying on large datasets of labeled samples, AL systems ask *queries* in form of unlabeled instances to be labeled by an *oracle* (e.g. a runtime experiment) [19]. This way, the active learner aims to achieve high accuracy while using as few labeled instances as possible, thereby minimizing the required benchmark runtime. Fuchs et al. show that AL facilitates significant speedups in the benchmarking process while achieving high ranking prediction accuracies[6].

1.1 Motivation

AL approaches rely on iteratively selecting samples for an *oracle* to label and updating the model based on the resulting label. As a result, they are inherently sequential. For practical use, this is not ideal, as it limits hardware scalability to the scalability of singular runtime experiments. In many cases, it would be preferred to run the benchmark experiments for multiple instances in parallel. This, however, limits the information available to the AL model when updating. We seek to expand the AL approach to incremental benchmark selection described by Fuchs et al. to facilitate parallel runtime experiments[6].

1.2 Contribution

In this thesis, we quantify the loss of accuracy through parallelization and explore and compare different approaches to mitigate it. We show that naive parallelization utilizing uncertainty sampling retains an advantage over sequential random sampling. We further propose a novel approach to parallelizing the oracle of an AL problem by treating its intermediate results as partial labels in an Active Partial Label Learning (APLL) problem. We propose and implement two methods of applying this APLL approach to SAT solver benchmarking. Evaluation on the SAT Competition 2022 Anniversary Track dataset shows that our geometric mean disambiguation method further improves the performance of naive parallelization. Geometric mean disambiguation can determine a new solver's PAR-2 ranking with about 91% accuracy while only needing 13.5% of the CPU-runtime of a full evaluation and running up to 64 benchmark experiments in parallel.

2 Preliminaries and Related Work

Our approach seeks to combine AL methods for incremental benchmark selection with Partial Label Learning methods to facilitate parallelization. In the following we will give an overview of related work in Benchmark Selection, Active Learning, Partial Label Learning and Distribution-based Runtime Predictions:

2.1 Benchmark Selection

Benchmarking is not only a vital part of many research processes, but has become a field of research in its own right. Recent studies highlight the challenges of benchmark selection, both in terms of efficiency and quality of results. A biased selection of benchmarks can easily skew the results and lead to false interpretations [4]. So can the multiplicity of design and analysis options combined with questionable research practice [17]. For SAT solver benchmarking, the problem of benchmark selection is well explored:

Static Benchmark Selection

The SAT Competitions hold annual international algorithm implementation contests. For ranking the solvers, organizers compose static benchmark instance sets. Balint et al. give an overview of common selection criteria in solver competition [1]. Frokley et al. provide a detailed description of benchmark selection in the 2020 competition [5]. Manthey et al. [14] as well as Mısır et al.[16] each provide feature-based approaches for reducing the amount of necessary benchmarks and removing redundancies. Hoos et al. discuss the composition of representative benchmark sets, suggesting *instance variety*, *adapted instance hardness* and *avoidance of duplicate instances* as selection criteria [8]. However, these approaches do not incorporate incremental benchmark selection and do not optimize benchmark runtime.

Incremental Benchmark Selection

Fuchs et al. as well as Matricon et al. describe incremental approaches for benchmark instance selection [6, 15]. Matricon et al. address the *per-set efficient algorithm selection problem* (PSEAS). Given a new solver and an already benchmarked one, they iteratively

select instances from a benchmark set for the new solver to determine if it performs better than the established one. They optimize their selection to achieve a required level of confidence while minimizing the computational resources required. While this problem is related to Problem 1, ranking solvers based on pair-wise comparison does not scale well for comparing a new solver to a solver portfolio.
Fuchs et al. apply an AL approach to the *New-Solver Problem* 1. They provide a framework for ranking a new solver with respect to a portfolio of solvers with complete benchmarks on a given benchmark set. On the SAT Competition 2022 Anniversary Track dataset, their framework reached about 92% ranking accuracy while only taking about 10% of the runtime summed over all benchmark instances. However, their approach lacks parallelization, selecting instances one at a time. Since we aim to expand their approach, we will give a more detailed overview of their work in 2.5.

2.2 Active Learning

Active Learning is a machine learning technique commonly employed in scenarios where acquiring labels is expensive and little to no labeled training data exists. The defining characteristic of AL models lies in their ability to select instances for an *oracle* to label. Common scenarios for their application include *membership query synthesis*, where the AL model can select existing instances as well as generate new *interesting* instances to label and *pool-based active learning*, where the model selects the most *interesting* instance from a pool of instances [19]. Both have applications in benchmark selection: While synthesis based methods within the field of SAT are used to generate benchmark instances that best differentiate given solvers [3, 7, 22], pool based active learning can be directly applied to Problem 1.
Different metrics exist for what constitutes an *interesting* instance. Common query strategies include aiming to optimize *information gain*, i.e., querying the instance that would induce the greatest change to the current model if its label was known [20] or to minimize *uncertainty* (or entropy), i.e., querying the instance for which the model is least certain of how to label it [21].

2.3 Distribution Based Runtime Predictions

Matricon et al. make the assumption that runtime features of an instance and instance feature vectors of other instances are both predictive of the runtimes of the new solver $\hat{a}$ on an instance [15]. For example, if all algorithms in $\mathcal{A}$ solve an instance very quickly, then so should $\hat{a}$. In other words, $\hat{a}$ is expected to have similar behavior as the algorithms in $\mathcal{A}$. Similarly, if two feature vectors x_{if} and $x_{i'f}$ are close for two instances $x_i, x_{i'}$, then their runtimes should be close as well. Similar assumptions are made by Hutter et al. and Kerschke et al. [10, 12]. Based on these assumptions, Matricon et al. predict the probability

distributions δ_{x_i} over $[0, \tau]$, expressing that $\delta_{x_i}(t)$ is the probability of $\hat{a}$ solving the instance x_i with runtime t. To obtain the distributions, they treat the timeouts as right-censored data points and correct them statistically before fitting a Cauchy distribution to the data using Maximum Likelihood Estimation. We utilize a similar assumption for predicting the label confidence on a partially labeled instance. Since we discretize the runtimes into runtime labels and reserve a separate label for timeouts, we do not need to treat them as censored data and can instead model them separately to later combine the distributions into a mixture model. Instead of using a parametric approach for modeling the distribution, we use a non-parametric estimation to better account for multimodal distributions.

2.4 Active Partial Label Learning

Partial Label Learning (PLL) describes a form of weakly supervised learning, where each training instance x_i is equipped with a set of candidate labels S_i, among which only one is the true label. Li et al. describe PLL problems in the context of Active Learning for cases with large amounts of unlabeled data and few labeled or partially labeled samples. They utilize a method proposed by Zhang et al. of iteratively disambiguating candidate label confidence using instance feature relations in a k-Nearest Neighbor approach [13, 24]. We interpret the partial runtimes of runtime experiments running in parallel as partial labels by discretizing their runtime distribution into cluster label confidences. To disambiguate these label confidences, we use the method proposed by Zhang et al. [24].

2.5 Active Learning for SAT Solver Benchmarking (Fuchs et al.)[6]

Framework

Given a set of solvers $\mathcal{A}$, instances $\mathcal{I}$ and runtimes r, they first initialize a prediction model $\mathcal{M}$ for the new solver $\hat{a} \notin \mathcal{A}$ (Algorithm 1, Line 1). The prediction model $\mathcal{M}$ is used to repeatedly select an instance (Algorithm 1, Line 4) for benchmarking $\hat{a}$ (Algorithm 1, Line 5). The acquired result is subsequently used to update the prediction model $\mathcal{M}$ (Algorithm 1, Line 7). When the stopping criterion is met (Algorithm 1, Line 3), they quit the benchmarking loop and predict the final score of $\hat{a}$ (Algorithm 1, Line 8). Algorithm 1 returns the predicted score of $\hat{a}$ as well as the acquired instances and runtime measurements (Algorithm 1, Line 9).

Runtime Transformation

Instead of predicting the runtime of an instance, Fuchs et al. predict its runtime cluster. This discretization has shown to increase prediction accuracy while only mildly decreasing

Algorithm 1: Incremental Benchmarking Framework (Fuchs et al.)

`Input`: Solvers $\mathcal{A}$, Instances $\mathcal{I}$, Runtimes $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$, Solver $\hat{a}$
`Output`: Predicted Score of $\hat{a}$, Measured Runtimes $\mathcal{R}$

1 $\mathcal{M} \leftarrow \text{initModel}(\mathcal{A}, \mathcal{I}, r)$
2 $\mathcal{R} \leftarrow \emptyset$
3 `while` not stop $(\mathcal{M})$ `do`
4 $\quad e \leftarrow \text{selectNextInstance}(\mathcal{M})$
5 $\quad t \leftarrow \text{runExperiment}(\hat{a}, e)$ *// Runs $\hat{a}$ on e with timeout τ*
6 $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{(e, t)\}$
7 $\quad \text{updateModel}(\mathcal{M}, \mathcal{R})$
8 $s_{\hat{a}} \leftarrow \text{predictScore}(\mathcal{M})$
9 `return` $(s_{\hat{a}}, \mathcal{R})$

ranking accuracy when comparing the label ranking to PAR-2 scoring. The real-valued runtimes are transformed into discrete runtime labels on a per-instance basis. For each instance $e \in \mathcal{I}$, a clustering algorithm is used to assign the runtimes in $\{r(a, e) | a \in \mathcal{A}\}$ to one of 3 clusters C_0, C_1, C_2 such that the fastest runtimes for the instance e are in cluster C_0 and the slowest are in cluster C_1. Timeouts τ form a separate cluster C_2. The runtime transformation function $\gamma_k : \mathcal{A} \times \mathcal{I} \to \{0, 1, 2\}$ is then specified as follows:

$$\gamma_k(a, e) = j \iff r(a, e) \in C_j \tag{2.5.1}$$

Given an instance $e \in \mathcal{I}$, a solver $a \in \mathcal{A}$ belongs to the $\gamma_k(a, e)$-fastest solvers on the instance e.

Solver Model

For the solver model, Fuchs et al. employ a stacking ensemble of two prediction models: A quadratic-discriminant analysis and a random forest with a simple decision tree deciding which of the two ensemble members makes the prediction on which instance. This model $\mathcal{M}$ takes the features as input and provides a function $f : \hat{\mathcal{A}} \times \mathcal{I} \to [0, 1]^3$ for all solvers $\hat{\mathcal{A}} := \mathcal{A} \cup \{\hat{a}\}$, predicting the probabilities for the 3 discrete runtime labels. The model is rebuilt in every iteration, since running the experiments (Algorithm 1, Line 5) dominates the runtime of model training by magnitudes.

Instance Selection

Fuchs et al. show that in general cases, a sampling strategy that minimizes uncertainty performs best. The model selects the unlabeled instance $e \in \mathcal{I} \setminus \tilde{\mathcal{I}}$ for labeling that lies

closest to its decision boundaries, i.e. that minimizes $U(e)$, which is specified as follows:

$$U(e) := \left| \frac{1}{k} - \max_{n \in \{1,\dots,k\}} f(\hat{a}, e)_n \right| \tag{2.5.2}$$

3 Active Partial Label Learning for Parallel Benchmarking

We seek to expand the incremental benchmark framework proposed by Fuchs et al. [6] to facilitate parallel runtime experiments. The core functionality is the same as in Fuchs et al.[6]. Given a set of solvers $\mathcal{A}$, instances $\mathcal{I}$ and runtimes r, we initialize a prediction model $\mathcal{M}$ for the new solver $\hat{a} \notin \mathcal{A}$ (Algorithm 2, Line 1). We use the prediction model $\mathcal{M}$ to select instances (Algorithm 3, 4, Line 3) for benchmarking $\hat{a}$ (Algorithm 3, Line 5; Algorithm 4, Line 6). The result is subsequently used to update the prediction model $\mathcal{M}$ (Algorithm 3, Line 8; Algorithm 4, Line 11). When the stopping criterion is met (Algorithm 3, 4, Line 2) we quit the benchmarking loops and predict the final score (Algorithm 2, Line 6).

Algorithm 2: Parallel Benchmarking Framework

`input` : Solvers $\mathcal{A}$, Instances $\mathcal{I}$, Known Runtimes $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$, Solver $\hat{a}$, Number of Threads n

`output`: Predicted Score of $\hat{a}$, Measured Runtimes $\mathcal{R}$

1 $\mathcal{M} \leftarrow \text{initModel}(\mathcal{A}, \mathcal{I}, r, \hat{a})$
2 $\mathcal{R} \leftarrow \emptyset$
3 $\mathcal{T} \leftarrow \emptyset$
4 `for` $i = 0$ *to* n `do`
5 `  spawn` $\text{experimentThread}(\mathcal{M}, \mathcal{A}, \mathcal{I}, r, \hat{a})$
6 $s_{\hat{a}} \leftarrow \text{predictScore}(\mathcal{M})$
7 `return` $(s_{\hat{a}}, \mathcal{R})$

3.1 Parallelization

We will not address parallelizing the model updates and predictions. On the one hand, it is not clear how to parallelize model updates; on the other hand, it does not promise significant runtime reductions in our scenario, since the runtime of the benchmark experiments dominate the model update time by magnitudes. Instead, we explore methods to parallelize the benchmark experiments while keeping the model updates and predictions sequential.

Algorithm 3: Naive Parallelization Experiment Thread

`input` : Model $\mathcal{M}$, Solvers $\mathcal{A}$, Observed Runtimes $\mathcal{T}$, Instances $\mathcal{I}$, Known Runtimes $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$, Solver $\hat{a}$

```
1 lock(M)
2 while not stop(M) do
3     e ← selectNextInstance(M)
4     unlock(M)
5     t ← runExperiments(â, e)
6     lock(M)
7     R ← R ∪ {(e, t)}
8     updateModel(M, R)
9 unlock(M)
```

Algorithm 4: Partial Label Parallelization Experiment Thread

`input` : Model $\mathcal{M}$, Solvers $\mathcal{A}$, Observed Runtimes $\mathcal{T}$, Instances $\mathcal{I}$, Known Runtimes $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$, Solver $\hat{a}$

```
1  lock(M, T)
2  while not stop(M) do
3      e ← selectNextInstance(M)
4      setTimestamp(T, e)
5      unlock(M, T)
6      t ← runExperiments(â, e)
7      lock(M, T)
8      R ← R ∪ {(e, t)}
9      R_p ← getPartialRuntimes(T)
10     L ← disambiguateLabels(A, I, r, R_p)
11     updateModel(M, R, L)
12 unlock(M, T)
```

We start n threads to run experiments in parallel (Algorithm 2, Line 5). We lock the model $\mathcal{M}$ to ensure sequential model updates (Algorithm 3, Lines 1, 4, 6, 9; Algorithm 4, Lines 1, 5, 7, 12) and exit the benchmarking loop once the stopping criterion is met (Algorithm 3, 4, Line 2). This way, parallelization only affects the amount of information that the model can access. In the following, we will explore two parallelization approaches:

3.1.1 Naive Parallelization

In a naive approach to parallelization described in Algorithm 3, we update the model sequentially after every completed benchmark experiment. We use the information of all available completed experiments to update the model. The information quantity available to the model when selecting the j-th instance for labeling can be described as

$$\Theta^j_{\text{NP}} = (j - n) \cdot \iota \tag{3.1.1}$$

where n is the amount of total experiment threads and ι is a unit of information corresponding to one instance label. In contrast, in the sequential approach the model has the information

$$\Theta^j_{\text{Seq}} = (j - 1) \cdot \iota \tag{3.1.2}$$

Consequently, we expect the predictions at framework runtime to be less accurate and therefore the selected instances to be less ideal for benchmarking. After the stopping criterion is met, we choose to finish the benchmark experiments that are still running in other threads. This way, under the assumption of a uniform stopping criterion, a sequential and a parallel approach have the same amount of information after all threads finish:

$$\Theta^J = J \cdot \iota \tag{3.1.3}$$

where J is the total number of labeled instances. The only difference for the final ranking prediction is the informational value of the instance labels. We will use this approach as a baseline in our experiments.

3.1.2 Parallelization utilizing Partial Runtimes

Running benchmark experiments in parallel offers another information resource that can be exploited to close the information gap compared to the sequential approach. The parallel experiments supply partial runtime information for their respective benchmark instances. Algorithm 4 describes an approach that utilizes this: Whenever a benchmark experiment finishes, the corresponding thread collects the partial runtimes from all other benchmark experiments running in parallel (Algorithm 4, Line 9). This can be realized with a shared data structure that is locked and unlocked along with the model (Algorithm 4, Lines 1, 5,

7, 12) and holds the starting timestamps of all running experiments (Algorithm 4, Line 4). With these partial runtimes, we estimate the most probable true label of the unfinished benchmark instances (Algorithm 4, Line 10). The model $\mathcal{M}$ is then updated utilizing this additional information alongside all information from the completed experiments (Algorithm 4, Line 11). This approach supplies the following information to the model:

$$\Theta^{j}_{\text{PRP}} = (j - n) \cdot \iota + (n - 1) \cdot \iota_p \tag{3.1.4}$$

with ι_p being a unit of information corresponding to a partial label. The information value of ι_p strongly depends on the quality of our true label estimation and might even be negative if incorrect estimates predominate. In the following, we describe our label disambiguation methods:

3.2 Label Disambiguation

In addition to the instance features and the instance labels resulting from the runtime experiments, our algorithm supplies a set of partial runtimes to the model. We treat an instance x_i for which only the partial runtime is provided as a partially labeled instance with the candidate label set S_i including all possible labels. To make this information usable for the prediction model $\mathcal{M}$, we need to determine the label $y_i \in S_i$ with the highest probability of being the true label of x_i. We evaluate two approaches for disambiguating the candidate labels of a partially labeled instance:

- Geometric Mean: This approach uses the runtime features of instance x_i for disambiguation.
- K-Nearest Neighbor Label Propagation: This approach uses the runtime and instance features of instance x_i as well as of all other labeled or partially labeled instances for disambiguation.

3.2.1 Geometric Mean

The partial runtime t_i of an instance x_i alone does not provide enough information to predict its true label. According to the assumption in 2.3, we can make a prediction based on t_i and the runtime features $x_{ir}, (1 \leq r \leq |\mathcal{A}|)$ of x_i which contain the runtimes of all solvers in $\mathcal{A}$ on x_i. To predict the label of an instance x_i, we compute the mean $\bar{x}_{ir}$ of its runtime features. Intuitively, this can be interpreted as the expected runtime of a new solver on x_i. We then determine the cluster c that contains $\bar{x}_{ir}$. The cluster label l_c serves as the true label $y_i \in S_i$ to train our model $\mathcal{M}$. We choose the geometric mean (GM) over the arithmetic mean, since the runtime data has a large range and variation and is often skewed or contains extreme outliers. To realize the prediction as a function of t, we exclude all runtime features from the calculation with a runtime lower than t. Therefore, the mean can

be computed as

$$\bar{x}_{ir}(t) := \exp\left(\frac{1}{n}\sum_{k=1}^{n}\log(x_k)\right) \quad \text{for} \quad x_k \in \{x > t | x \in x_{ir}\} \tag{3.2.1}$$

where n is the number of runtime features larger than t.

3.2.2 K-Nearest Neighbor Label Propagation

Using the geometric mean to estimate the label based on the partial runtime of an unfinished instance has some important drawbacks. One disadvantage of this method is that it does not consider the confidence in its label prediction.

For a partially labeled instance (x_i, S_i), it returns $y_i \in S_i$ with $\bar{x}_i \in C_{y_i}$, which is then treated as a correctly labeled instance by the prediction model $\mathcal{M}$, regardless of whether the confidence in the label prediction approaches 1 or barely exceeds that of the second most probable label.

To implement a prediction model that incorporates and maximizes label confidence, we first need to devise a method to obtain the label confidence of each label in the candidate label set S_i from the partial runtime of an instance and its runtime features:

Label Confidence

We derive the label confidence of all candidate labels S_i from the runtime features and the observed partial runtime $t_i \in \mathcal{R}_p$ of an instance x_i. For an instance x_i with the runtime features $x_{ir}, (1 \leq r \leq |\mathcal{A}|)$, we can fit a probability density function (PDF) to the distribution of x_{ir}. The integral of the PDF between two cluster boundaries can be interpreted as the confidence of the cluster label being the true label $y_i \in S_i$ for $t = 0$. To decide on a model for computing the PDF, we make the following observations:

(i) The amount of data is limited to $|\mathcal{A}|$

(ii) When visually inspecting the runtime features x_{ir} of randomly chosen instances, we can observe unimodal as well as multimodal distributions 3.1.

(iii) For runtime features containing timeouts, these timeout features are grouped at the timeout penalty value 2τ with zero variance.

To accommodate a flexible shape of the distribution and fine-grained control on smoothness, we choose gaussian Kernel Density Estimation (KDE):

$$\hat{f}_i(x) = \frac{1}{nh}\sum_{r=1}^{n} K\big(\frac{x - x_{ir}}{h}\big), \qquad K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}} \tag{3.2.2}$$

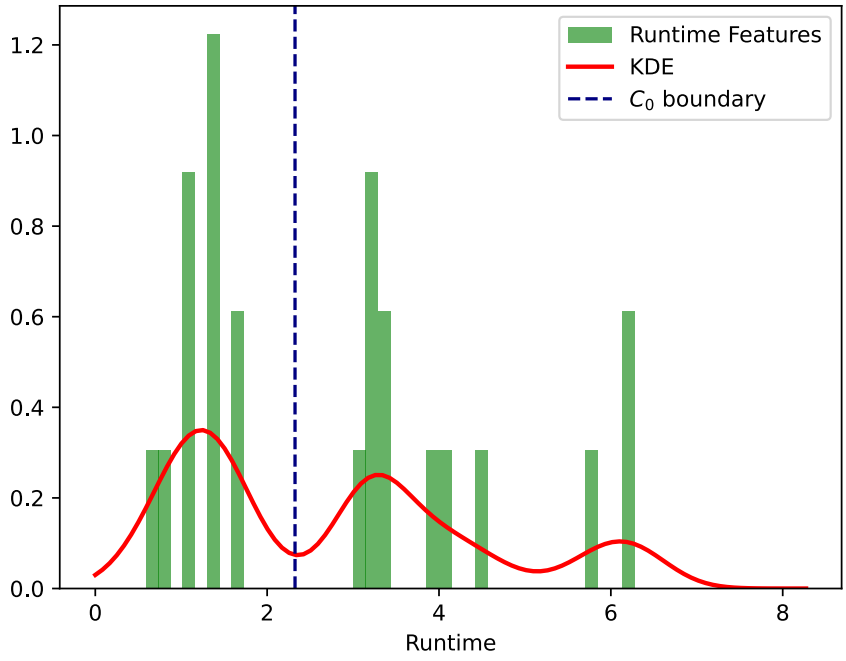

Figure 3.1: Runtime features of instance "08cb0cf252b65c30c7c55218825c6ca0", $\hat{f}_i(x)$ with $h = 0.25$ and the upper cluster boundary of C_0. This instance displays a trimodal distribution.

where n is the number of runtime features, h is the bandwidth (smoothing parameter), x_r are the runtime features and K is the gaussian kernel with $u = \frac{x - x_{ir}}{h}$ representing the standardized distance between the point x and each sample point x_{ir}.

The timeout features are independent of the non-timeout runtime features. To account for their zero variance, we model them separately with a Dirac Delta Function $\delta(x)$:

$$\delta_i(x) = \begin{cases} 0 & x \neq 0 \\ \infty & x = 0 \end{cases}, \text{ so that } \int_{-\infty}^{\infty} \delta(x) = 1 \tag{3.2.3}$$

We combine the KDE (3.2.2) and the Dirac Delta Function (3.2.3) into a mixture model to create a PDF for all runtime features x_{ir} of an instance x_i:

$$f_i(x) = (1 - \omega) \cdot \hat{f}_i(x) + \omega \cdot \delta_i(x - 2\tau) \tag{3.2.4}$$

where 2τ equals the timeout penalty value and ω represents the weight of the Dirac Delta Function and equals the fraction of timeouts in the runtime features x_{ir}. We can now interpret the integral of $f_i(x)$ between the boundaries c_{low} and c_{high} of a cluster c as the

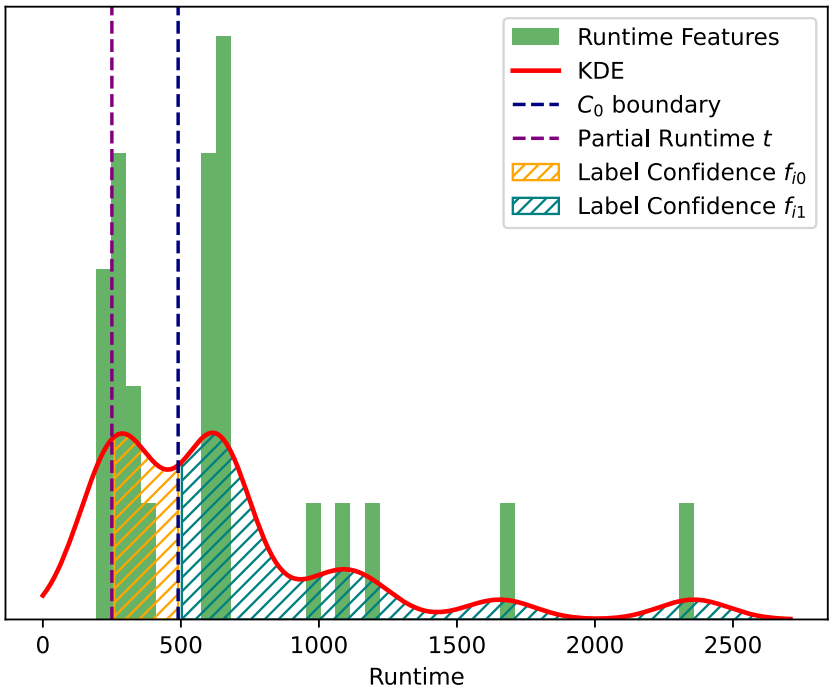

Figure 3.2: Runtime features, KDE and upper cluster boundary C_0 of instance "0041051c73dcdd885d412a38e8b09fba". The areas under the curve show the label confidence f_{i0} and f_{i1} with respect to partial runtime t, assuming the KDE is scaled to $\int_t^\infty \text{KDE} = 1$ and there are no timeouts.

probability f_{il_c} of the cluster label l_c being the true label of x_i. But so far, this does not incorporate the partial runtime information t_i. To describe cluster label probabilities as a function of t, we scale the PDF so that $\int_t^\infty f_i(x) = 1$. For a cluster c with c_{low} and c_{high} as lower and upper boundaries and $t < c_{\text{high}}$, the cluster label probability $f_{il_c}(t)$ can be described as:

$$f_{il_c}(t) := \int_{c_{\text{low}}}^{c_{\text{high}}} f_i^{(t)}(x) = \frac{1}{1-(1-\omega)\int_{-\infty}^{t}\hat{f}_i(x)} \cdot \int_{\max(t,c_{\text{low}})}^{c_{\text{high}}} (1-\omega)\cdot\hat{f}(x) + \omega\cdot\delta(x-2\tau) \quad (3.2.5)$$

We define $f_{il_c}(t)$ to be the label confidence in label l_c at the partial runtime t. By calculating $f_{il_c}(t_i)$ for all $l \in S_i$, we can construct a confidence vector $F_i = [f_{i1}, f_{i2}, f_{i3}]$ for each instance x_i in respect to its partial runtime $t_i \in \mathcal{R}_p$. This denotes our confidence in label l being the true label of instance x_i, where $f_{il} \in [0,1]$ and $\sum_{l=1}^{q} f_{il} = 1$. Then the confidence matrix of candidate labels for our partially labeled instances is $F_p = [f_1, f_2, \ldots, f_p]^\top$. In

the following we will disambiguate the confidence matrix leveraging the affinity between the instances derived from their features.

K-Nearest Neighbors

Another drawback of the geometric mean approach is that it relies solely on the runtime features of the particular instance. Our experiments have shown that this approach is not sufficient to enhance the prediction model accuracy. To improve upon this method, we utilize the method described by Zhang et al. [24] to incorporate the runtime and instance features of the other labeled and partially labeled instances as well as the instance features of x_i into the true label estimation:
For each instance $(x_i, S_i) \in D_p$, let $N(x_i)$ denote the indexes of its k-nearest neighbors in D_p. We determine a similarity matrix $W = [w_{i,j}]_{p \times p}$ where $w_{i,j} = 0$ if $x_j \notin N(x_i)$. For $x_{j_a} \in N(x_i), 1 \leq a \leq k$, let $w_i = [w_{i,j_1}, w_{i,j_2}, \ldots, w_{i,j_k}]^\top$ be the weight vector of x_i and its k-nearest neighbors in W. The influence of each neighbor x_{j_a} on x_i can then be computed by solving optimization problem (3.2.6) as follows:

$$\begin{aligned} &\min_{w_i} \left\| x_i - \sum_{a=1}^{k} w_{i,j_a} \cdot x_{j_a} \right\| \\ &\text{s.t. } w_{i,j_a} \geq 0 (x_{j_a} \in N(x_i), 1 \leq a \leq k) \end{aligned} \tag{3.2.6}$$

As shown in (3.2.6), the weight vector w_i is optimized by fitting a linear least square problem subject to non-negativity constraints. The optimal solution $\hat{w}_i$ can be obtained by applying any off-the-shelf quadratic programming solver. Intuitively, the magnitudes of the weights $\hat{w}_i$ now encode the strength of affinity between an instance and its k nearest neighbors in D_p.

Label Propagation

We now leverage this information about the affinity between instances to refine label confidence. In an iterative process, the confidence vectors of partially labeled instances are influenced by the confidence vectors of other partially and fully labeled instances to which they are neighbored. We normalize the weight matrix W by row as $H = D^{-1}W$, where $D = \text{diag}[d_1, d_2, \ldots, d_p]$ is a diagonal matrix and $d_j = \sum_{i=1}^{p} w_{i,j}$. The confidence matrix F_p is now iteratively updated based on the label propagation and the normalized weight matrix H. In the η-th iteration, $\tilde{F}_p^{(\eta)}$ is computed as:

$$\tilde{F}_p^{(\eta)} = \alpha \cdot H^\top F_p^{(\eta-1)} + (1 - \alpha) \cdot F_p^{(0)} \tag{3.2.7}$$

where $\alpha \in (0, 1)$ controls the relative amount of information inherited from label propagation and the initial label confidence. For the candidate label set of each instance, $\tilde{F}^{(\eta)}$ is

rescaled as $F^{(\eta)}$:

$$\forall 1 \leq i \leq p : f_{il}^{(\eta)} = \frac{\tilde{f}_{il}^{(\eta)}}{\sum_{y_{l'} \in S_i} \tilde{f}_{il'}^{(\eta)}} \tag{3.2.8}$$

When the iteration stops, each partially labeled instance (x_i, S_i) can be disambiguated by the final obtained confidence matrix $\hat{F}$:

$$\hat{y}_i = \underset{y_l \in S_i}{\operatorname{argmax}} \hat{f}_{il} \tag{3.2.9}$$

The disambiguated labels are then used to update the prediction model $\mathcal{M}$.

4 Experimental Setup

This section outlines our experimental design, including our evaluation framework, the used datasets and hyperparameters.

4.1 Implementation

To evaluate Algorithm 2, we simulate the parallel algorithm in a sequential evaluation framework 5. For all specific parameter instances and sub-routines, we perform cross-validation on our set of solvers, with each solver acting as the *new* solver $\hat{a}$ once (Line 2). To avoid data leakage, the new solver is removed from the set of solvers $\mathcal{A}$ in each iteration (Line 3). The only segment in Algorithm 2 that is not locked and therefore runs in parallel is the runtime experiment (Algorithm 2, Line 14). We simulate this with a list of runtimes, including the partial runtimes from unfinished runtime experiments (Line 4). Additionally, we track for each selected instance whether it is running or finished (Line 8, 16, 20, 24). This way we can distinguish partial runtimes from final runtimes when computing the label confidence (Line 13, 15).
Before starting the benchmarking loop, we select n instances and set them to *running*. This corresponds to spawning n threads in algorithm 2 (Algorithm 2, Line 5). We then enter the benchmarking loop. We find the running instance e' with the shortest remaining runtime $rt_{e'}$ (Line 10). This is possible because we selected the *new* solver $\hat{a}$ from the test data and already know its true runtimes on all instances. By only using this information for the instance that is set to *finished* (Line 15) before the next update of the prediction model $\mathcal{M}$ (Line 17), we ensure that no data is leaked to the model. We increment the partial runtimes of all *running* instances by $rt_{e'}$ (Line 13) and set instance e' to *finished*. We calculate the label confidence from the partial runtimes of all *running* instances (Line 14). To decide on a true label estimate for training $\mathcal{M}$, we then disambiguate the label confidence matrix (Line 16). We update the model (Line 18) and select a new instance $\hat{e}$ which we set to *running*.
After the stopping criterion is reached (Line 9), we let the remaining experiments finish (Line 21–23), update the model (Line 24) and estimate the final score $s_{\hat{a}}$ (Line 25). Like in [6], we then calculate the ranking accuracy and the runtime of $s_{\hat{a}}$. We define the *ranking accuracy* $O_{\text{acc}} \in [0, 1]$ (higher is better) by the fraction of pairs $(\hat{a}, a)$ for all $a \in \mathcal{A}$ that are decided correctly regarding the ground-truth scoring par_2 (Lines 26–28). The *fraction of runtime* $O_{rt} \in [0, 1]$ (lower is better) puts the summed runtimes of the sampled instances

Algorithm 5: Implementation of Evaluation Framework

`input` : Solvers $\mathcal{A}$, Instances $\mathcal{I}$, Runtimes $r : \mathcal{A} \times \mathcal{I} \to [0, \tau]$, Number of Threads n
`output`: Average Ranking Accuracy $\bar{O}_{\text{acc}}$

$O \leftarrow \emptyset$
`for` $\hat{a} \in \mathcal{A}$ `do`
  $\mathcal{A}' \leftarrow \mathcal{A} \setminus \{\hat{a}\}, \quad \mathcal{R} \leftarrow \emptyset$
  `repeat` n `times`
    $e \leftarrow$ selectNextInstance($\mathcal{M}$)
    set status of e to running
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(e, 0)\}$
  `while` ***not*** stop `do`
    find $e' \in \mathcal{R}$ ***with*** status$(e) =$ running ***and shortest remaining runtime*** $rt_{e'}$
    $\mathcal{F}_p \leftarrow \emptyset$
    `for` $(e, r) \in \mathcal{R}$ ***where*** status$(e) =$ running `do`
      $r \leftarrow r + rt_{e'}$
      $\mathcal{F}_p \leftarrow \mathcal{F}_p \cup$ getLabelConfidence(e, r)
    set status ***of*** e' ***to*** finished
    $\mathcal{F}_p \leftarrow$ disambiguatePartialLabels$(\mathcal{F}_p)$
    updateModel$(\mathcal{M}, \mathcal{R}, \mathcal{F}_p)$
    $\hat{e} \leftarrow$ selectNextInstance$(\mathcal{M})$
    set status ***of*** $\hat{e}$ ***to*** running
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(\hat{e}, 0)\}$
  `for` $(e, r) \in \mathcal{R}$ ***where*** status$(e) =$ running `do`
    $r \leftarrow r + rt_e$
    set status ***of*** e ***to*** finished
  updateModel$(\mathcal{M}, \mathcal{R})$
  $s_{\hat{a}} \leftarrow$ predictScore$(\mathcal{M})$
  $O_{acc} \leftarrow 0$ `for` $a \in \mathcal{A}$ `do`
    `if` $\left(s_k(a) - s_{\hat{a}}\right) \cdot \left(\texttt{par}_2(a) - \texttt{par}_2(\hat{a})\right) > 0$ `then`
      $O_{acc} \leftarrow O_{acc} + \frac{1}{|\mathcal{A}|}$
  $r \leftarrow \sum_{e \in \mathcal{I}} r(\hat{a}, e), \quad O_{rt} \leftarrow 0$
  `for` $e \in \mathcal{I}$ `do`
    `if` $\exists t, (e, t) \in \mathcal{R}$ `then`
      $O_{rt} \leftarrow O_{rt} + \frac{t}{r}$
  $O \leftarrow O \cup \left\{(O_{acc}, O_{rt})\right\}$
$(\bar{O}_{\text{acc}}, \bar{O}_{\text{rt}}) \leftarrow$ average(O)
`return` $(\bar{O}_{\text{acc}}, \bar{O}_{\text{rt}})$

in relation to the runtime summed over all instances in the dataset. Finally, we compute averages of the output metrics (Line 30) from all cross-validation results.

4.1.1 Hyper-Parameters

Given our Evaluation Framework 5, we now evaluate several methods for *parallelization*. Additionally, there are different choices of methods and hyperparameters for *sampling* and *stopping*. We describe these configurations in the following.

Selection.

For selection, we use the following methods and hyperparameter values:

- Random Sampling
- Uncertainty Sampling (2.5.2)
  - Fallback threshold: Use random sampling for the first $0\%, 1\%, 2\%$ of instances.

We choose uncertainty sampling since [6] suggests this query strategy to perform best in most cases. Furthermore, we use random sampling as a baseline for comparison.

Parallelization.

We evaluate the following parallelization methods and hyperparameter values:

- Naive Parallelization (3.1.1)
  - Number of Threads: Execute $1, 2, 4, 8, 16, 32, 64$ benchmark experiments in parallel
- Partial Runtimes with Geometric Mean Label Estimation (3.2.1)
  - Number of Threads: Execute $1, 2, 4, 8, 16, 32, 64$ benchmark experiments in parallel
- Partial Runtimes with k-Nearest Neighbor Label Estimation (3.2.2)
  - Bandwidth: h of $0.25, 0.5$. Representing prioritizing sensitivity or smoothness of the KDE, respectively.
  - Number of Threads: Execute $1, 2, 4, 8, 16, 32, 64$ benchmark experiments in parallel.
  - Number of Neighbors: Consider $5, 10, 15$ nearest neighbors for partial label disambiguation.
  - Label Propagation Importance: Relative information gain of $\alpha \in \{0.25, 0.5, 0.6, 0.7, 0.8, 0.9\}$ from label propagation as opposed to initial label confidence.
  - Amount of label propagation iterations: 50.

We aim to compare the performance of the different parallelization approaches across a varying number of threads to assess their applicability to real world benchmarking problems. For the *k-nearest neighbor* based label prediction approach, we additionally evaluate a set of hyperparameters.

Stopping.

For stopping decisions, we use the subset-size criterion to stop after sampling $2.5\%, 5\%, 7.5\%, 10\%, 12.5\%, 15\%, 17.5\%, 20\%, 30\%, 40\%$ of instances. Thereby, the different parallelization approaches and baseline experiments sample a similar quantity of instances. This allows for easier comparison between approaches.

4.1.2 Environment

Our code is implemented in PYTHON using *gbd-tools* [11] for retrieving SAT-instances, *scikit-learn* [18] for the prediction model and computing the nearest neighbors and *scipy* for computing the Gaussian kernel-density-estimation and solving the optimization problem. For reproducibility, our source code and data are available on *GitLab*[1].

4.1.3 Data

To facilitate comparability to the results of Fuchs et al. [6], we use the SAT Competition 2022 Anniversary Track dataset in our experiments [2]. The dataset consists of 5355 instances with respective runtime data of 28 sequential SAT solvers. Furthermore, we use a database of 56 instance features[2] from the Global Benchmark Database (GBD) by Iser et al. [11]. Among others, they comprise instance size features and node distribution statistics for several graph representations of SAT instances and are primarily inspired by the SATzilla 2012 features described in [23]. We discard 10 out of 56 features because of zero variance. Overall, the prediction models have access to 46 instance features and 27 runtime features of 5355 instances, as one out of the 28 solvers acts as the new solver at any time. For hyperparameter tuning, we create a smaller instance dataset by randomly sampling 535 instances.

[1] `https://gitlab.kit.edu/ubfcl/active-parallel-partial-label-learning-for-efficient-benchmarking`

[2] `https://benchmark-database.de/getdatabase/base_db`

5 Experimental Evaluation

In this section, we evaluate the different parallelization approaches. We first analyze and tune the hyperparameters for the *k-nearest neighbor* based label prediction approach, and then evaluate all approaches on the complete dataset. Each configuration is repeated 5 times with different random seeds. The random sampling baseline experiments are repeated 10 times instead. All O_{rt}-O_{acc}-diagrams in this chapter show Pareto fronts. We do not show intermediate results of the evaluation framework. Instead, evaluating a specific configuration with Algorithm 5 returns a point $(O_{\text{rt}},O_{\text{acc}})$. The plotted lines represent the maximum ranking accuracy with the displayed model and parameter fixed and all other parameters variable.

5.1 Hyper-Parameters

For KNN disambiguation, we evaluate the hyperparameters α and k on the test dataset. We perform a full evaluation with all combinations of the parameter values described in 4.1.1, $4, 8$ and 32 threads and sampled runtime fractions of $0.05, 0.1, 0.15, 0.2$ and 0.3. To analyze the parameters, we compute the mean label ranking accuracy for fixed parameter values over all other parameters, respectively.
Both parameters display a low sensitivity with regard to the label ranking accuracy. As shown in Figure 5.1, we can observe a peak in the hyperparameter evaluation of α. We therefore choose $\alpha = 0.8$ for evaluating KNN disambiguation on the full dataset. Interestingly, this value differs from the α value of 0.95 suggested in [24], which indicates that the initial label confidence matrix in our approach provides additional useful information in contrast to the initial matrix in Zhang et al., which starts with evenly distributed initial confidence.
For evaluation on the full dataset, we further choose $k = 10$, as we find no meaningful sensitivity to this parameter and this value is suggested by [24].

5.2 Comparing the Approaches

In the following, we compare the different parallelization approaches to each other, as well as to a sequential random sampling and an uncertainty sampling baseline.

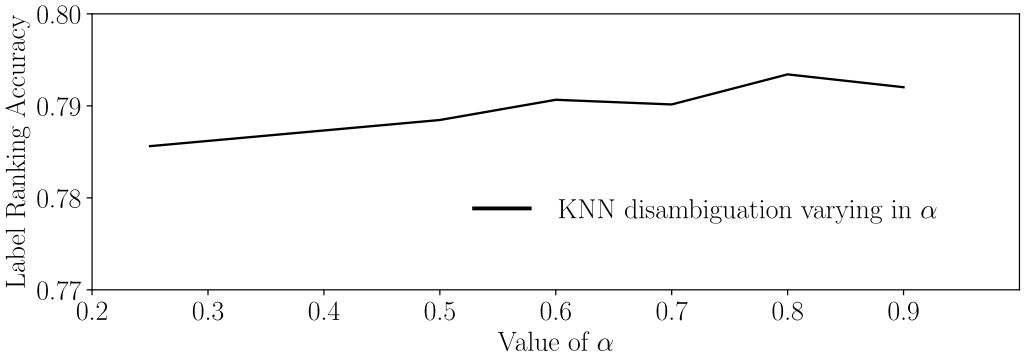

Figure 5.1: Diagrams showing the relation between the values of hyperparameter and label ranking accuracy on the test dataset.

5.2.1 Naive Parallelization and Geometric Mean Disambiguation

Across all tested numbers of threads, both naive parallelization and geometric mean disambiguation perform strictly better than the random sampling baseline 5.2. This outcome is expected, as [6] has shown AL to outperform random sampling in a sequential approach. We therefore expect that it retains at least some advantage with limited information.
With an increasing number of threads, the geometric mean disambiguation approach increasingly outperforms the naive parallelization. With 64 threads, the best performing configuration of GM disambiguation predicts the PAR-2 rank of a new solver with 89.52% accuracy while only running benchmark experiments that account for 8.12% of the total benchmark set CPU-runtime. This proves that the label estimations made by the geometric mean disambiguation on average provide useful information to the model.

5.2.2 KNN Disambiguation

When evaluated on the full dataset, KNN disambiguation consistently performs worse than naive parallelization 5.2. With increasing number of threads, it even falls below the random sampling baseline. This behavior is unexpected. There are a number of possible interpretations of these results:

(i) KNN disambiguation does not scale well to large datasets.

(ii) One or more hyperparameters are sensitive to the dataset size.

(iii) An error in the implementation of the evaluation framework (5) distorts results on large datasets.

In the following, we examine each hypothesis.

(i)

The large difference in the resulting accuracy between evaluating KNN disambiguation on the full dataset and the test dataset could indicate that the method does not scale well to large datasets. However, experiments conducted by Zhang et al. on multiple datasets of varying sizes do not show comparable behavior [24]. Our approach only differs from Zhang et al. in the modified initial label confidence matrix. The computation of the confidence matrix is independent of the dataset size.

(ii)

Since we tune the hyperparameters k and α on the test dataset, the performance on the full dataset could be significantly worse if one or both of these parameters are sensitive to the dataset size. However, in the parameter sensitivity analysis performed by Zhang et al. no sensitivity with regard to the dataset size is found [24]. Furthermore, our experiments with the hyperparameters on the test dataset observed only a low sensitivity of the ranking accuracy with respect to changes in k and α.

(iii)

Neither hypothesis (i) nor hypothesis (ii) account for the ranking accuracy to drop below the accuracy of random sampling. This observation suggests that the average value of a unit of information ι_p (3.1.4) provided by KNN disambiguation on the full dataset is in fact negative. I.e. the amount of false predictions outweighs any information gain from AL. The fact that this only occurs on the full dataset suggests an error in the implementation of KNN disambiguation in the evaluation framework that is sensitive to dataset size.
Further analysis is required to draw an unambiguous conclusion about the performance of KNN disambiguation.

5.2.3 Thread Scalability

For different amounts of threads, GM disambiguation shows very similar ranking accuracies 5.3. For larger number of threads, the observed ranking accuracy is slightly higher, especially around a sampled runtime fraction of 0.1. This result is unexpected, as we assume the ranking accuracy to decrease with an increasing fraction of partial labels in the model training data. The uncertainty sampling baseline also yields lower ranking accuracy than expected. We assume the uncertainty sampling baseline to outperform any configuration of the parallelization approaches.
However, these observations can be attributed to statistical variation. The best configuration of a sequential approach with uncertainty sampling described by Fuchs et al. achieves an 92.33% accuracy with a sampled runtime fraction of 0.104. This outperforms the best

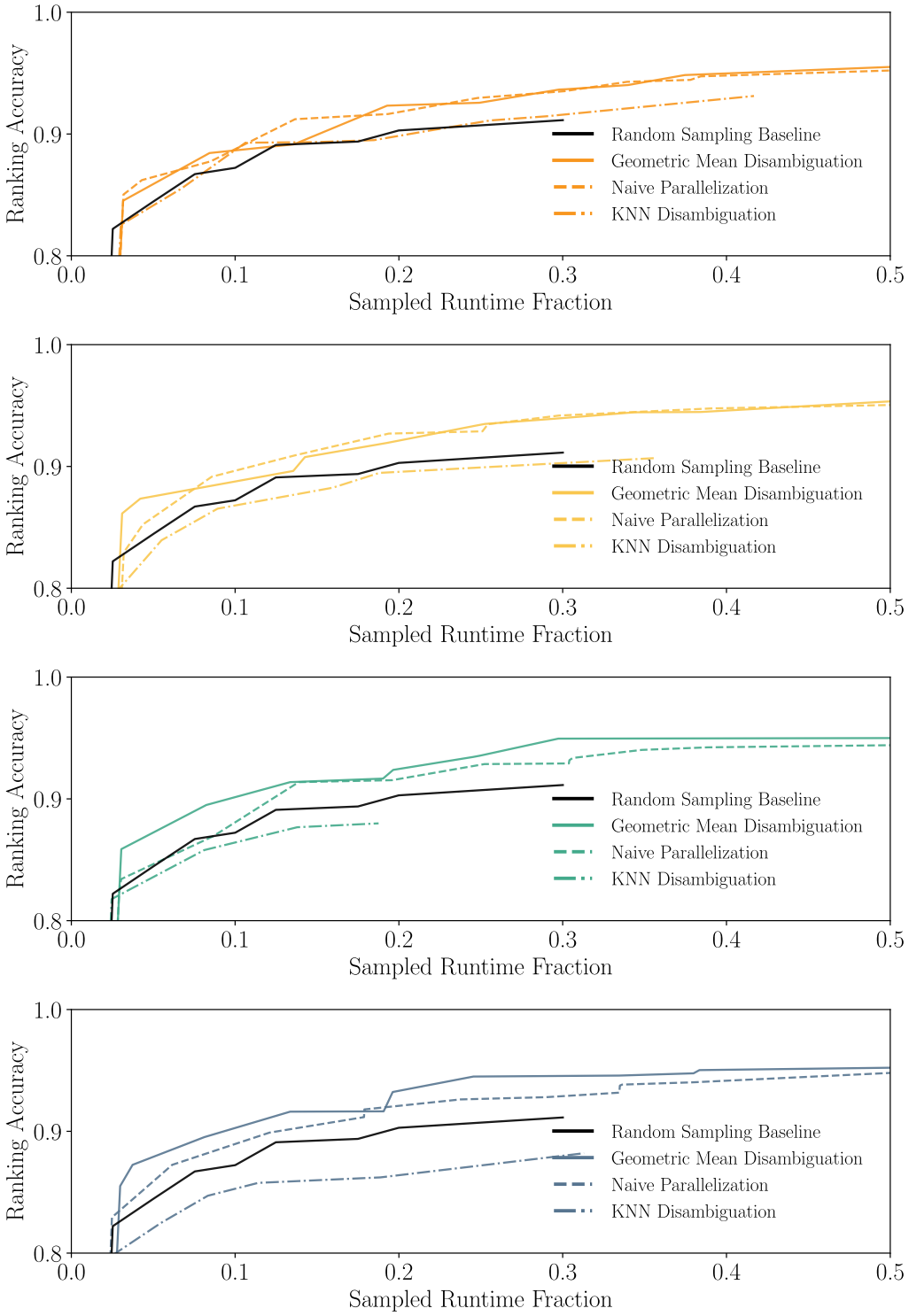

Figure 5.2: O_{rt}—O_{acc} diagrams comparing the Pareto fronts of naive parallelization, geometric mean and k-nearest neighbor disambiguation for $4, 8, 32$ and 64 threads respectively on the full dataset. The x-axis shows the ratio of total solver CPU-runtime on the sampled instances relative to the solver CPU-runtime on all instances in the benchmark set. The y-axis shows the accuracy of the predicted PAR-2 ranking.

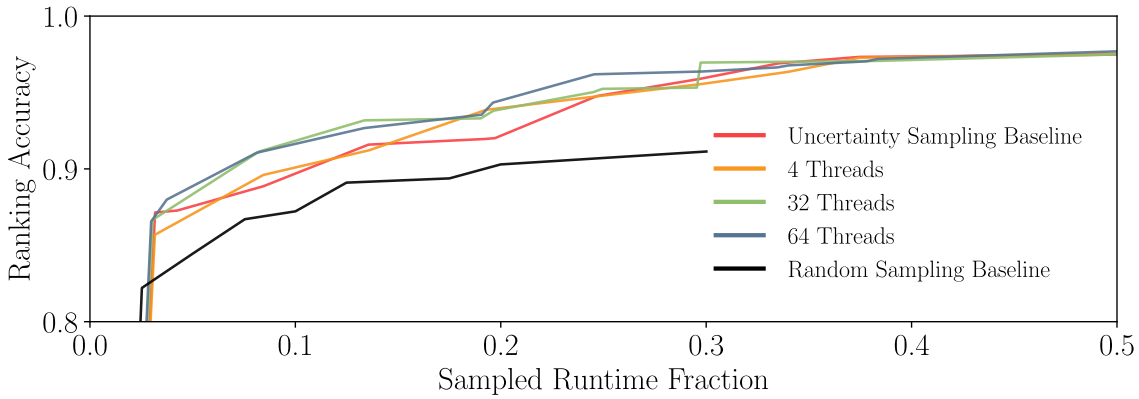

Figure 5.3: O_{rt}-O_{acc} diagram comparing the Pareto fronts of GM disambiguation on the full dataset for $4, 32$ and 64 threads, as well as the random sampling and uncertainty sampling baseline.

performing configuration of GM disambiguation (32 threads: 91.38% ranking accuracy over 0.134 sampled runtime fraction) as expected.

6 Conclusion and Future Work

In this thesis, we explore parallelizable approaches to the *New-Solver Problem* 1. We establish a naive parallelization method as a baseline approach and show that it retains an advantage over random sampling. We further improve the approach by applying principles of Partial Label Learning to the runtime data of unfinished benchmark experiments. On data from the SAT Competition 2022 Anniversary Track, we can determine a new solver's PAR-2 ranking with about 91% accuracy while only needing 13.5% of the CPU-runtime of a full evaluation and running up to 64 benchmark experiments in parallel by using *geometric mean* disambiguation. We also examine *k-nearest neighbor* as another PLL method to disambiguate partial runtime data. However, we did not reach an unambiguous conclusion regarding its performance.

Future work may further analyze the *k-nearest neighbor* approach on the SAT Competition 2022 Anniversary Track to achieve comparable results. From a technical perspective, one can further investigate the scalability of our approaches regarding the number of experiment threads. Additionally, one can further investigate the stopping criterion. We employ the subset size criterion for stopping to facilitate comparability, however this might not be ideal for real world application. The optimal stopping criterion for sequential approaches uses ranking convergence, but this may be sensitive to false label predictions by the PLL disambiguation. Further improvements to the stopping criterion for both sequential and parallel approaches may be possible by terminating slow instances early if further runtime data does not add valuable information to the model. One approach to this is treating the terminated runtimes as right censored observations [9]. Improvements to the selection criterion may also be possible. Li et al. describe a selection strategy for PLL methods that incorporates graph density and label propagation ability in addition to conditional entropy [13].

Furthermore, one can apply the concept of treating intermediate results of an AL oracle as partial labels to facilitate oracle parallelization to other AL problems.

Bibliography

[1] Adrian Balint et al. “Overview and analysis of the SAT Challenge 2012 solver competition”. In: *Artificial Intelligence* 223 (2015), pp. 120–155. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2015.01.002. URL: https://www.sciencedirect.com/science/article/pii/S0004370215000120.

[2] Tomas Balyo et al., eds. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. English. Department of Computer Science Series of Publications B. Finland: Department of Computer Science, University of Helsinki, 2022.

[3] Nguyen Dang et al. “A Framework for Generating Informative Benchmark Instances”. In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Ed. by Christine Solnon. Vol. 235. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 18:1–18:18. ISBN: 978-3-95977-240-2. DOI: 10.4230/LIPIcs.CP.2022.18. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2022.18.

[4] Mostafa Dehghani et al. “The Benchmark Lottery”. In: *CoRR* abs/2107.07002 (2021). arXiv: 2107.07002. URL: https://arxiv.org/abs/2107.07002.

[5] Nils Froleyks et al. “SAT Competition 2020”. In: *Artificial Intelligence* 301 (2021), p. 103572. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2021.103572. URL: https://www.sciencedirect.com/science/article/pii/S0004370221001235.

[6] Tobias Fuchs, Jakob Bach, and Markus Iser. “Active Learning for SAT Solver Benchmarking”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Cham: Springer Nature Switzerland, 2023, pp. 407–425. ISBN: 978-3-031-30823-9.

[7] Iván Garzón, Pablo Mesejo, and Jesús Giráldez-Cru. “On the Performance of Deep Generative Models of Realistic SAT Instances”. In: *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Ed. by Kuldeep S. Meel and Ofer Strichman. Vol. 236. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 3:1–3:19. ISBN: 978-3-95977-242-6. DOI: 10.4230/LIPIcs.SAT.2022.3. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2022.3.

[8] Holger H. Hoos et al. “Robust Benchmark Set Selection for Boolean Constraint Solvers”. In: *Learning and Intelligent Optimization*. Ed. by Giuseppe Nicosia and Panos Pardalos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 138–152. ISBN: 978-3-642-44973-4.

[9] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. “Bayesian Optimization With Censored Response Data”. In: (Oct. 2013). URL: https://www.cs.ubc.ca/~kevinlb/papers/2013-BayesianOptimizationCensored.pdf.

[10] Frank Hutter et al. “Algorithm runtime prediction: Methods evaluation”. In: *Artificial Intelligence* 206 (2014), pp. 79–111. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2013.10.003. URL: https://www.sciencedirect.com/science/article/pii/S0004370213001082.

[11] Markus Iser and Carsten Sinz. “A Problem Meta-Data Library for Research in SAT”. In: *Proceedings of Pragmatics of SAT 2015 and 2018*. Ed. by Daniel Le Berre and Matti Järvisalo. Vol. 59. EPiC Series in Computing. EasyChair, 2019, pp. 144–152. DOI: 10.29007/gdbb. URL: /publications/paper/jQXv.

[12] Pascal Kerschke et al. “Automated Algorithm Selection: Survey and Perspectives”. In: *Evolutionary Computation* 27.1 (Mar. 2019), pp. 3–45. ISSN: 1063-6560. DOI: 10.1162/evco_a_00242. eprint: https://direct.mit.edu/evco/article-pdf/27/1/3/1552398/evco_a_00242.pdf. URL: https://doi.org/10.1162/evco%5C_a%5C_00242.

[13] Yan Li et al. “Active partial label learning based on adaptive sample selection”. In: *International Journal of Machine Learning and Cybernetics* 13 (2022), pp. 1603–1617. URL: https://api.semanticscholar.org/CorpusID:246338084.

[14] Norbert Manthey and Sibylle Möhle. “Better Evaluations by Analyzing Benchmark Structure”. In: 2016. URL: https://api.semanticscholar.org/CorpusID:201024718.

[15] Théo Matricon et al. “Statistical Comparison of Algorithm Performance Through Instance Selection”. In: *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Ed. by Laurent D. Michel. Vol. 210. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 43:1–43:21. ISBN: 978-3-95977-211-2. DOI: 10.4230/LIPIcs.CP.2021.43. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2021.43.

[16] Mustafa Mısır. “Benchmark Set Reduction for Cheap Empirical Algorithmic Studies”. In: *2021 IEEE Congress on Evolutionary Computation (CEC)*. 2021, pp. 871–877. DOI: 10.1109/CEC45853.2021.9505012.

[17] Christina Nießl et al. “Over-optimism in benchmark studies and the multiplicity of design and analysis options when interpreting their results”. In: *WIREs Data Mining and Knowledge Discovery* 12.2 (2022), e1441. DOI: https://doi.org/10.1002/widm.1441. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1441. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1441.

[18] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

[19] Burr Settles. *Active Learning Literature Survey*. Tech. rep. University of Wisconsin-Madison, Department of Computer Sciences, 2009. URL: http://digital.library.wisc.edu/1793/60660.

[20] Burr Settles, Mark W. Craven, and Soumya Ray. “Multiple-Instance Active Learning”. In: *Neural Information Processing Systems*. 2007. URL: https://api.semanticscholar.org/CorpusID:12956.

[21] C. E. Shannon. “A Mathematical Theory of Communication”. In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: https://doi.org/10.1002/j.1538-7305.1948.tb01338.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1948.tb01338.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1948.tb01338.x.

[22] Toan Tran et al. “Bayesian Generative Active Deep Learning”. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 6295–6304. URL: https://proceedings.mlr.press/v97/tran19a.html.

[23] Lin Xu et al. *Features for SAT*. Tech. rep. University of British Columbia, 2012. URL: https://www.cs.ubc.ca/labs/beta/Projects/SATzilla/Report_SAT_features.pdf.

[24] Min-Ling Zhang and Fei Yu. “Solving the Partial Label Learning Problem: An Instance-Based Approach”. In: *International Joint Conference on Artificial Intelligence*. 2015. URL: https://api.semanticscholar.org/CorpusID:209702.