Bachelor's Thesis

Mapping the Hyperparameter Space Using Performance-Influence Models

Lukas Klein

August 5, 2025

Advisor:
Kallistos Weis Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Jilles Vreeken CISPA Helmholtz Center for Information Security

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Hyperparameter optimization is a key factor for learning high performing machine learning models. However, it is very challenging due to the large number of potential configurations. This complexity not only increases computational cost but also makes it hard to interpret how individual hyperparameters affect model performance.

Configurable software systems similarly deal with a huge search space of valid configurations and the challenge of understanding how feature interactions impact the overall system performance. This observation raises the question of whether well-established approaches from this domain can be effectively applied to the challenges of hyperparameter optimization.

Therefore, our research investigates the use of sampling-based performance-influence models to accurately predict machine learning model performance across the full hyperparameter search space. Our findings demonstrate that sampling a very small subset of all configurations using diversified distance-based and random sampling, yields performance-influence models with low prediction errors in most cases and successfully identifies top-performing hyperparameter settings comparable to Hyperopt-sklearn.

Acknowledgments

I am deeply thankful for the support I received throughout the process of writing this thesis. First and foremost, I would like to express my gratitude to my advisor, Kallistos Weis, for his guidance and support throughout my work. I would also like to thank Sebastian Böhm for always being available to answer questions regarding the implementation of my experiment. Furthermore, my thanks go to Prof. Dr. Sven Apel for the opportunity to write my thesis at his chair.

And finally, I could not have undertaken this journey without my family and friends. Thank you for your support, encouragement, and patience throughout the writing process.

Contents

List of Figures

List of Tables

Listings

Acronyms

AI	Artificial Intelligence
ML	Machine Learning
TPE	Tree-structured Parzen Estimators
FM	Feature Model
PIM	Performance-Influence Model
MNB	Multinomial Naive Bayes
DTC	Decision Tree Classifier
GBC	Gradient Boosting Classifier
SGD	Stochastic Gradient Descent
KNN	K-Nearest Neighbors
RFC	Random Forest Classifier

1 Introduction

In recent years, the relevance of Artificial Intelligence (AI) has grown continuously. Whether it's speech recognition, disease diagnosis, self-driving cars, generative AI, or large language models like ChatGPT, advances in AI are becoming more prominent in everyone's daily life. All these systems are built on complex, resource-intensive Machine Learning (ML) models and optimizing them typically involves maximizing performance while minimizing computational costs. A crucial part of developing high-performing ML models is hyperparameter tuning. Hyperparameters (i.e., settings that control the learning process) play a major role in determining both the performance and overall quality of an ML model. Here, the main challenge is the large number of possible hyperparameter combinations. Exhaustively testing every combination is inefficient and oftentimes even infeasible due to the combinatorial explosion of the search space [1].

To address this challenge, automated ML tools such as Hyperopt-sklearn have been developed [9]. These tools automate the hyperparameter optimization process, which allows even non-experts to efficiently tune models without exploring the entire search space manually. AutoML tools simplify the ML pipeline and the development of high-quality ML models [13]. Current state-of-the-art approaches in automated hyperparameter optimization use advanced ML techniques like Bayesian optimization to navigate the hyperparameter search space [14, 25]. For example, Hyperopt-sklearn uses Tree-structured Parzen Estimators (TPE). This model guides the search towards the most promising regions of the search space, thereby improving optimization efficiency compared to exhaustive search methods. However, Bayesian techniques tend to be highly complex and are often difficult to understand and interpret. This complexity can be problematic when the goal is not only to find optimal hyperparameters but also to understand the external behavior and the inner workings of the system.

In the domain of configurable software systems, similar challenges of searching large search spaces are addressed using sampling techniques. These systems use Performance-Influence Models (PIMs) that are learned from a subset of configurations. Once trained, these models can predict the performance across the entire configuration space, thus reducing the need for exhaustive evaluation [22].

The primary goal of this thesis is to investigate and assess the use of PIMs for predicting the performance of different hyperparameter configurations of ML models (RQ1). A further objective is to evaluate whether this sampling-based approach can be used for hyperparameter optimization, in comparison to a state-of-the-art Bayesian-based optimization tool, namely Hyperopt-sklearn (RQ2). Specifically, we aim to determine whether PIMs can serve as an alternative that is not only accurate but also interpretable. To achieve this, we

represent an ML model as a configurable system by treating the model's hyperparameters as its features. In particular, we use *sampling* (e.g., diversified distance-based and random sampling) to create a representative subset of the hyperparameter space and *ML techniques* (e.g., linear regression) to learn a PIM. To generate the ground-truth performance measurements required both for training our PIMs and for assessing their prediction accuracy, we will use six scikit-learn classifiers. In the final step, we compare the prediction results from our approach with the optimization results produced by Hyperopt-sklearn. In contrast to Hyperopt-sklearn, which concentrates on identifying *the* optimal configuration, our focus is on achieving accurate performance predictions across the entire configuration space.

We show that sampling just 1–10% of each classifier's hyperparameter space and fitting stepwise linear regression PIMs, we achieve prediction errors below 1% for Multinomial Naive Bayes, Decision Tree, K-Nearest Neighbors and Random Forest, and 2–5% for Gradient Boosting, while Stochastic Gradient Descent performed worst. Using the top-3 configurations predicted by these PIMs, we find hyperparameter configurations that are competitive to the ones found by Hyperopt-sklearn on five of six classifiers, with improvements from $+0.03$ to $+1.20$ pp accuracy, though we underperform on stochastic gradient descent by -0.52 pp. Moreover, our approach yields a global performance map of the entire hyperparameter landscape, providing both competitive tuning and full interpretability.

2

Background

In this chapter, we present important background information to understand and follow the contents of this thesis. We discuss configurable software systems, feature models, performance-influence model, hyperparameter optimization and Hyperopt-sklearn for AutoML.

2.1 Configurable Software Systems

Configurable software systems are designed to offer multiple features, which makes them adaptable to different requirements and user preferences. The main goal of configurable systems is to support variability and allow customization of software behavior. The difference between individual software variants lies in the selection or deselection of specific features. In this context, features refer to units of functionality that can either be included or excluded in a given software variant. This flexibility is important in domains such as product line engineering, where a single core product is modified to produce multiple software variants based on feature selections [2, 6].

In configurable software systems, features can be classified into three types: binary, numerical and categorical. Binary features are the simplest, as they involve a straightforward inclusion or exclusion decision. For instance, enabling a feature means the associated functionality will be included in the software variant, while disabling it implies the opposite. For example, a software product may have a *Dark Mode* feature that is either enabled or disabled. Numerical features involve selecting a specific value from a predefined range, i.e. a non-binary decision. For example, the amount of RAM allocated to a software instance could be modelled as a numerical feature with a value space of *2GB*, *4GB*, *8GB* and *16GB*. On the other hand, categorical features can take on multiple discrete values, where each value represents a different category. For example, a software product may offer different database backends such as *MySQL*, *PostgreSQL* and *MongoDB*. Here, the feature *Database Type* is categorical because it can take one of several predefined values.

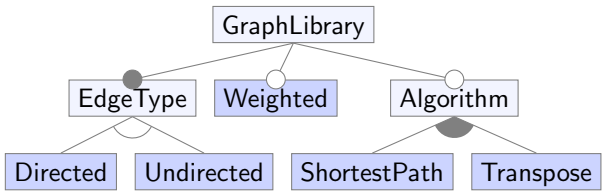
The configuration space C is defined by the Cartesian product of all possible features. However, due to constraints such as dependencies between features, logical contradictions and hardware limitations, not every configuration in C is valid. In particular, these constraints are Boolean expressions that define whether a configuration is valid or not, i.e., which features are allowed in which combinations [12]. By applying these constraints, we obtain the valid configuration space $V \subseteq C$, which is a subset of C containing only valid configurations.

However, configurable software systems are highly complex. The valid configuration space V expands exponentially with the number of optional and independent features [2], and each configuration $c \in V$ can potentially lead to other system behavior and performance. As a result, making use of a simple brute-force approach to analyze performance becomes infeasible for most larger software systems. Therefore, it is important to develop efficient methods for understanding how configuration options and their interactions affect performance.

2.2 Feature Models

Following Apel et al. [2], a Feature Model (FM) defines all valid combinations of features within the system. There are different ways of defining a FM, but we will focus on the tree-like representation. This tree structure includes mandatory and optional features, where each feature is linked through a parent-child relationship. Parent-child relations define the dependency hierarchy of features, while or-groups (at least one child is selected) and xor-groups (exactly one child is selected) specify how features can be grouped or selected together. Additionally, FMs can make use of cross-tree constraints, which define dependencies or restrictions across different branches of the tree, in order to ensure that the model accurately reflects the allowed configurations of the system.

Example 1. A FM representing a configurable graph library.



$Transpose \Rightarrow Directed$

Example 1 showcases the FM of a simple configurable graph library. For instance, the feature *Edge Type* is mandatory and its children are grouped in an xor-group. These constraints lead to the fact that each configuration is either a directed or undirected graph. On the other hand, the feature *Algorithm* is optional and its children are grouped as an or-group, which indicates that if a configuration supports algorithms, it can include *ShortestPath* or *Transpose*, or both. $Transpose \implies Directed$ is an example of a cross-tree constraint which defines that if *Transpose* is selected, also *Directed* has to be selected.

2.3 Performance-Influence Models

A Performance-Influence Model (PIM) is designed to describe how features and their interactions influence the performance of a system. The objectives of PIMs are understanding, optimization, and debugging of highly configurable software systems. An end-user may use an optimizer to find the best performing configuration under certain constraints from the

model. On the other hand, developers can compare the PIM with their own mental model of the system behavior [22]. Such comparisons can be helpful to validate whether the system behaves as intended or to detect potential performance bugs. An ideal PIM should have high prediction accuracy, short computation time and small model size. Usually, there are tradeoffs between these properties that do not allow to optimize for all of them at once. However, Kolesnikov et al. [15] state that accurate PIMs can be learned in feasible time. Thus, a PIM might be an efficient alternative to more complex and costly optimization and prediction approaches.

Siegmund et al. [22] define PIMs formally as follows: Assume that F represents the set of all features and V symbolizes the set of all valid configurations. Each configuration $c \in V$ can be modeled as a function $c : F \to \mathbb{R}$ that assigns a selected value to each feature: For a binary feature f, $c(f) = 1$ if the option is selected and $c(f) = 0$ if it is not selected. For a numerical feature f, $c(f)$ returns a number in the value range of that feature.

Definition 1. A PIM is a function that maps configurations to a performance metric $\Pi : V \to \mathbb{R}$. This function is represented by:

$$\Pi(c) = \beta_0 + \sum_{i \in F} \phi_i(c(i)) + \sum_{i..j \in F} \Phi_{i..j}(c(i)..c(j))$$

In simpler terms, Definition 1 is a sum of the constant base performance, the sum of the influences of all individual features and the sum of the influences of all interactions among all features.

Example 2.

$$\Pi(c) = 50 + 15 \cdot c(D) - 5 \cdot c(U) + 8 \cdot c(W) + 4 \cdot c(S) + 2 \cdot c(T) - 0.5 \cdot c(U) \cdot c(S) + 12 \cdot c(D) \cdot c(W) \cdot c(S)$$

Example 2 showcases the PIM of a simple configurable graph library. It consists of five options: $Directed(D)$, $Undirected(U)$, $Weighted(W)$, $ShortestPath(S)$ and $Transpose(T)$. For instance, 50 describes the constant base performance. $15 \cdot c(D)$ describes the performance influence of the option D and $-0.5 \cdot c(U) \cdot c(S)$ describes the performance influence of the feature interaction between U and S.

2.4 Hyperparameter Optimization

In ML, hyperparameters are key settings which control the learning process, such as model structure, training dynamics and regularization. Well-tuned hyperparameters can improve the performance of a model, while badly chosen ones can lead to suboptimal results. A Ilemobayo et al. [1] emphasize that hyperparameter tuning is a critical factor that affects the ML performance, alongside the data quality, the choice of algorithm, and the model complexity. Specific hyperparameters, like *learning rate* and *batch size*, can have a strong impact on these properties.

Both hyperparameters in ML and features in configurable software systems act as external "knobs" that can be tuned to shape the behavior and performance of the model and software system, respectively. They are manually chosen or optimized through optimization algorithms. This external configuration layer provides flexibility, allowing both data scientists and software engineers to tailor their systems to meet specific requirements, optimize performance and adapt to varying contexts. A Ilemobayo et al. [1] cover a range of methods for hyperparameter tuning:

Grid Search explores all possible combinations of hyperparameters in a structured, evenly spaced grid, which can provide optimal results but quickly becomes computationally intensive in high-dimensional spaces.

Random Search randomly selects hyperparameter combinations. This technique is more computationally efficient than grid search, by exploring a broader set of possibilities without the cost of an exhaustive search.

Bayesian optimization uses probabilistic models to iteratively select the most promising hyperparameters, balancing exploration and exploitation, which allows it to find optimal values with fewer evaluations. Classical Bayesian optimization is based on gaussian processes and is considered to be well-suited for continuous search spaces. A popular modification of Bayesian optimization is *TPE* which is based on density estimation and is preferred for discrete search spaces [21, 24].

2.5 AutoML: Hyperopt-sklearn

AutoML strives to make ML easier for non-experts by automating the end-to-end ML pipeline, i.e., performing common data science tasks with minimal effort for the human. A very important component of this pipeline are *AutoMHL frameworks* which automate the model selection and hyperparameter tuning process. By exploring different combinations, these frameworks search for the best model and parameters for a given dataset [4].

Hyperopt is a popular example of such a framework [3]. It is a python library designed for hyperparameter optimization and uses TPE to explore complex search spaces efficiently. It is oftentimes integrated with other AutoML frameworks.

Scikit-learn, also called *sklearn*, is an open-source ML library in Python, which is built on top of NumPy, SciPy and Matplotlib [19]. Sklearn provides supervised and unsupervised learning algorithms (e.g., classification and regression) while keeping it easy and efficient to use.

Hyperopt-sklearn is an extension of scikit-learn that introduces automated hyperparameter optimization to the model selection process, by incorporating the Hyperopt library [9, 16]. Hyperopt-sklearn treats the selection of classifiers, regressors and preprocessing methods as a single large optimization problem. Instead of manually trying different models and tuning hyperparameters, it automates the process using Hyperopt's optimization algorithms. This way one can achieve high accuracy without extensive manual experimentation.

The first step when using Hyperopt-sklearn is to define a search space over all possible configurations of scikit-learn components, including *preprocessing techniques* such as normalization, PCA or TF-IDF, *classification models* like K-Nearest Neighbors, SVM or Random

Forest, and *regression models* such as Linear Regression. After establishing the search space, Hyperopt explores it through three main components: the *search domain*, which includes the full range of hyperparameters and model choices; the *objective function*, which evaluates each model configuration according to a chosen performance metric such as accuracy or F1-score; and the *optimization algorithm*, which guides the selection of the most promising hyperparameter configurations, whether using TPE, grid search or random search.

Example 3. Hyperopt-sklearn follows a sklearn-like API. The following showcases the usage of this python library:

Listing 2.1: Python source code example for Hyperopt-sklearn

```
from hpsklearn import HyperoptEstimator, random_forest_classifier
from hyperopt import hp, tpe

# Define the estimator:
estim = HyperoptEstimator(
    algo=tpe.suggest, # Define the optimization algorithm (here we use TPE)
    max_evals=100,
    trial_timeout=600,
    # In this example, we only use random forest with a limited search space for
    # n_estimators and max_depth:
    classifier=random_forest_classifier('clf',
        n_estimators=hp.choice('n_estimators', [10, 50, 100]),
        max_depth=hp.choice('max_depth', [None, 5, 10, 20]),
    ),
    # Optionally, you can add preprocessing steps here:
    preprocessing=[],
    seed=42
)
# Train the estimator:
estim.fit(train_data, train_label)

# Make predictions:
predictions = estim.predict(test_data)

# Get the best model found:
best_model = estim.best_model()
```

3 Related Work

In this chapter, we present literature and approaches that are related to our work. In particular, we discuss the performance prediction pipeline, common sampling techniques, and different approaches to the learning process of performance-influence model.

3.1 Performance Prediction Pipeline

The Interplay of Sampling and Machine Learning for Software Performance Prediction

Kaltenecker et al. [11] present a four-step performance prediction pipeline: First, a sampling strategy selects a representative yet manageable subset of configurations from the enormous configuration space. Next, these chosen configurations are empirically measured to obtain their performance characteristics. Next, an ML algorithm learns a PIM from the collected performance measurements. Finally, the PIM is used to predict the performance of new, unmeasured configurations. The prediction accuracy can be assessed by comparing predictions to actual measurements.

The choice of the *sampling strategy* and the *ML method* have a significant impact on training cost and prediction accuracy [11]. As different systems have different performance implications, varying in linearity, interaction effects and noise levels, there does not exist one combination that is universally optimal. Thus, the techniques used for *sampling* and *learning* have to be chosen wisely.

3.2 Sampling Strategies

A key challenge in the performance analysis of highly configurable software systems is that the number of software variants grows exponentially in the number of features. Measuring the performance of each individual variant is infeasible [22, 23]. Therefore, the development of efficient sampling strategies is important, in order to select a representative subset of variants that still provide reliable performance measurements.

Distance-Based Sampling of Software Configuration Spaces

Kaltenecker et al. [12] provide an overview of common sampling methods: *Random sampling* is a simple sampling technique that tries to cover the configuration space uniformly [12, 17]. As the name suggests, it works by randomly selecting features from the configuration

space to create a valid configuration. For smaller projects with little to no constraints, this might be a good option. However, in practice, this technique does not scale, as configurable software systems usually are highly constrained. Thus, *random sampling* would produce invalid configurations most of the time and is therefore considered to be inefficient and impractical.

Solver-based sampling makes use of constraint solvers, such as SAT solvers, to efficiently search for valid configurations. However, this approach often results in a locally clustered set of configurations. In order to solve this problem, Henard et al. [8] present a *randomized solver-based* approach.

Coverage-based sampling strategies optimize the sample set according to a specific coverage criterion. A common method is t-wise sampling, where every combination of t features is guaranteed to appear in at least one configuration [10].

Distance-based sampling spreads configurations according to a given probability distribution [12]. It uses a distance metric (e.g., the Manhattan distance) to measure the number of selected features in each configuration. Using a discrete probability distribution, configurations with different distances are chosen, which leads to a better coverage of the configuration space. This method helps to capture various interaction patterns between features that affect performance. In order to further improve the variety within the set of samples, Kaltenecker et al. [12] extend this strategy by diversity optimization. This *diversified distance-based sampling* ensures that configurations are selected more evenly, which leads to even better prediction accuracy and reliability.

3.3 Learning Performance-Influence Models

Performance-Influence Models for Highly Configurable Systems

Siegmund et al. [22] propose a method for learning interpretable PIMs that explain how individual features and their interactions affect system performance. This transparency can be useful for debugging, performance optimization and understanding the overall system behavior.

In order to learn the function of a PIM from a sample set of measured configurations, Siegmund et al. [22] use *linear regression*. By mapping configuration values to performance measurements, linear regression estimates coefficients for both individual and interaction terms, which results in a clear understanding of performance influences.

Given that a complete model could contain an exponential number of potential terms, the paper introduces a *stepwise feature selection*[1] algorithm. First, there is a *forward feature selection* step and they start by initializing an empty feature set. Then the candidate features are evaluated, and the candidate term that minimizes the prediction error is added. After the model is built up, there is the *backward feature selection* step. Here, each feature is tested for whether its removal would decrease the accuracy of the model, which ensures that the final model remains compact and interpretable.

1 Note that in the context of ML, feature selection has nothing to do with features in configurable software systems.

Since exhaustively considering all possible terms is infeasible, the method relies on heuristics to select these candidates that are most likey to influence performance. For example, the algorithm assumes that interactions are hierarchical (i.e., first identify significant individual effects before considering their interactions).

Predicting Performance via Automated Feature-Interaction Detection

Siegmund et al. [23] present an approach which automatically detects and quantifies performance-relevant feature interactions. Their method builds on the insight that the influence of a feature on performance can vary significantly depending on the presence of other features.

To capture this, Siegmund et al. [23] introduce the concept of *performance deltas*, which quantify the impact of a feature by comparing the performance of configurations with and without the feature in question. By analyzing differences in deltas across various base configurations, the method identifies features that likely participate in interactions.

To manage the otherwise intractable number of possible feature combinations, the approach employs heuristics to narrow the search space: *pair-wise interaction detection*, which assumes that most relevant interactions are between two features; a *composition heuristic* for identifying higher-order interactions; and the notion of *hot-spot features* that frequently participate in interactions. This targeted strategy reduces the number of required measurements while maintaining high predictive accuracy.

Cost-Efficient Sampling for Performance Prediction of Configurable Systems

Sarkar et al. [20] combine a cost-aware sampling framework and regression tree modeling to efficiently learn PIMs. Rather than selecting a fixed-size sample in advance, Sarkar et al. [20] adapt two dynamic sampling strategies from the data mining domain, which are *progressive sampling* and *projective sampling*, to iteratively determine the optimal number of configuration measurements.

Both strategies are integrated with *Classification and Regression Trees (CART)*, to learn the performance model. The core idea is to balance the trade-off between prediction accuracy and measurement cost through a sampling cost model that incorporates the number of configurations sampled, the accuracy of the resulting model and the associated overhead. This sampling-guided learning framework results in cost-efficient and scalable performance predictions.

DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network

Ha and Zhang [7] present a deep learning approach to performance prediction for configurable software systems by modeling performance as a non-linear function of configuration options.

Their method, *DeepPerf*, makes use of a *deep feedforward neural network (FNN)* equipped with *L1 sparsity regularization* to construct a PIM from a limited set of sampled configurations. The deep FNN architecture allows for modeling complex interactions among features, while the L1 regularization ensures sparsity in the network weights—effectively filtering out

irrelevant interactions and improving generalizability. To handle the challenges of limited data and high-dimensional configuration spaces, Ha and Zhang [7] introduce a tailored hyperparameter tuning strategy that efficiently searches for suitable network structures and learning parameters.

Unlike earlier approaches that focus on interpretability, DeepPerf emphasizes prediction accuracy, especially for systems with both binary and numeric features. DeepPerf consistently achieves higher accuracy with fewer measurements compared to prior methods, especially in environments where performance-relevant feature interactions are sparse and non-linear [7].

4

Methodology

In this chapter, we present the methodology of the thesis. In particular, we discuss our research questions, our experiment, and our evaluation pipeline.

4.1 Research Questions

Configurable software systems have shown that by sampling a small, well-chosen subset of configurations and building a PIM, one can both predict behavior across the entire configuration space and gain interpretable insight into feature effects. In ML, by contrast, hyperparameter optimization typically uses Bayesian black-box methods (e.g., Hyperopt-sklearn) that excel at finding a single good setting but offer little transparency about why certain hyperparameters matter.

In our research, we apply the performance-influence modeling method to ML hyperparameter tuning: we treat scikit-learn classifiers as configurable systems (hyperparameters = features), sample a small fraction of the hyperparameter space, and learn PIMs via stepwise linear regression. Then, we use those PIMs to answer the following two research questions:

> **Research Question 1:** How accurately can performance-influence models predict the performance of machine learning models across the entire hyperparameter space?

First, we investigate how precisely PIMs can predict the accuracy of different ML models across the entire hyperparameter space compared to ground-truth performance measurements provided by scikit-learn.

To answer our first research question, we learn PIMs to predict the performance of all configurations within the respective hyperparameter search spaces. Each prediction is computed by evaluating the trained regression model on the corresponding hyperparameter configuration, resulting in an estimated performance value. We assess the prediction accuracy by comparing these estimated values against the ground-truth performance measurements. Furthermore, we want to find out to what extent the chosen sampling techniques (random sampling and diversified distance-based sampling) and the chosen number of sampled configurations (1%, 5%, and 10% of the total amount of configurations) have an impact on the performance prediction results.

> **Research Question 2:** How does our performance-influence model-based approach for hyperparameter optimization compare to Bayesian-based methods such as Hyperopt-sklearn in terms of accuracy?

State-of-the-art hyperparameter optimization methods like Hyperopt-sklearn use acquisition functions to find good settings quickly, but their models are opaque and only guide the search, not explain it. A PIM is fully interpretable, since each coefficient tells you how a hyperparameter or interaction affects performance. If it can also serve as a competitive hyperparameter optimization driver, we get both interpretability and optimization.

To answer our second research question, we compare our PIM-based approach against Hyperopt-sklearn by evaluating whether we can identify a hyperparameter configuration of comparable quality just by using our performance predictions. For each classifier, we select the PIM variant that achieved the lowest prediction error in the first research question (i.e., using 1%, 5%, or 10% of the hyperparameter space, depending on which sampling rate yielded the best model). To perform the comparison, we use Hyperopt-sklearn's TPE as the underlying search algorithm. We fix *max_evals* to 10%, the timeout on each optimization run to 60 seconds, and the random seed to 42, in order to ensure reproducible results. Finally, we compare the measured accuracy of the best three hyperparameter configurations suggested by our PIM-guided selection to the best one found by Hyperopt-sklearn.

4.2 Experiment

To answer our research questions, we will conduct an experiment for which we consider six scikit-learn classifiers; Multinomial Naive Bayes (MNB), Decision Tree Classifier (DTC), Gradient Boosting Classifier (GBC), Stochastic Gradient Descent (SGD), K-Nearest Neighbors (KNN), and Random Forest Classifier (RFC) (see Table 4.1). Each classifier will run through an evaluation pipeline, as we will describe in Section 4.3.

Table 4.1: Overview of the *classifiers* and their *number of features* ($|F|$) and *number of valid hyperparameter configurations* ($|V|$). We will use the abbreviations of the classifiers for the rest of the thesis.

Classifier	**|F|**	**|V|**
Multinomial Naive Bayes (MNB)	9	20
Decision Tree (DTC)	13	24
Gradient Boosting (GBC)	21	288
Stochastic Gradient Descent (SGD)	19	896
K-Nearest Neighbors (KNN)	30	960
Random Forest (RFC)	26	1 296

4.2.1 Data Set

For our experiments, we use the MNIST dataset [5], which is a standard benchmark for handwritten-digit classification. MNIST consists of 70 000 grayscale images of digits 0-9 (see Figure 4.1), split into 60 000 training images and 10 000 test images. Each image is 28×28 pixels, with intensity values in the range $[0, 255]$. MNIST's popularity and widespread use in research make it a suitable benchmark for our experiments. We are dealing with a typical classification task: Classify an image of a handwritten digit into the classes 0-9.

Figure 4.1: Extract from the MNIST dataset.

4.2.2 Performance Metrics

In this section, we introduce the metrics used for evaluating the classification accuracy of the sklearn classifiers, the prediction error of our PIMs, and the quality of our hyperparameter optimization outcomes.

Measuring the Classification Accuracy

Definition 2. Measured Accuracy
For each configuration $c \in V$, accuracy is defined as the proportion of correctly classified samples:

$$a_c = \frac{\text{Number of correctly classified test samples}}{\text{Total number of test samples}} \tag{4.1}$$

This *measured accuracy* a_c indicates how well each scikit-learn classifier configuration performs on held-out test data (see Section 4.2.1 for more details on the data set). The set of all measured accuracies is denoted as A — this is the *"true accuracy"*.

Evaluating the Prediction Accuracy (RQ1)

Definition 3. Predicted Accuracy
When evaluating our PIMs, we compare each $\hat{a}_c$, i.e., the predicted accuracy for configuration c, against its respective a_c value. The set of all predicted accuracies is denoted as $\hat{A}$.

Definition 4. Mean
The mean of the true accuracies $\bar{A}$ (also called the *true mean*) is defined as:

$$\bar{A} = \frac{1}{|V|} \sum_{c \in V} a_c, \tag{4.2}$$

Definition 5. Median
Let A be a sorted list of accuracies in non-decreasing order. Then, the median of the true accuracies ($\text{med}(A)$) is defined as:

$$\text{med}(A) = \begin{cases} \frac{1}{2}(a_{|V|/2} + a_{|V|/2+1}), & \text{if } |V| \text{ is even,} \\ a_{(|V|+1)/2}, & \text{if } |V| \text{ is odd.} \end{cases} \tag{4.3}$$

Definition 6. Variance and Standard Deviation
Variance and standard deviation are both measures of how spread out or dispersed a set of numbers is. The variance is the average of the squared differences from the mean and the standard deviation is the square root of the variance:

$$\sigma^2(A) = \frac{1}{|V|} \sum_{c \in V} (a_c - \bar{A})^2, \tag{4.4}$$

$$\sigma(A) = \sqrt{\sigma^2(A)}. \tag{4.5}$$

Definition 7. Mean Absolute Percentage Error
The Mean Absolute Percentage Error (MAPE) quantifies the average relative prediction error as a percentage, comparing predicted accuracies $\hat{a}_c$ to true accuracies a_c over all configurations $c \in V$:

$$\text{MAPE} = \frac{100\%}{|V|} \sum_{c \in V} \left| \frac{a_c - \hat{a}_c}{a_c} \right| \tag{4.6}$$

Evaluating our Predictions against Hyperopt-sklearn (RQ2)

To compare our approach against Hyperopt-sklearn, we take the top-3 configurations by predicted accuracy and look at their true accuracies. Then, we compare the best accuracy of those three to the optimum configuration found by Hyperopt:

Definition 8. Top-3 Prediction vs. Hyperopt Optimum

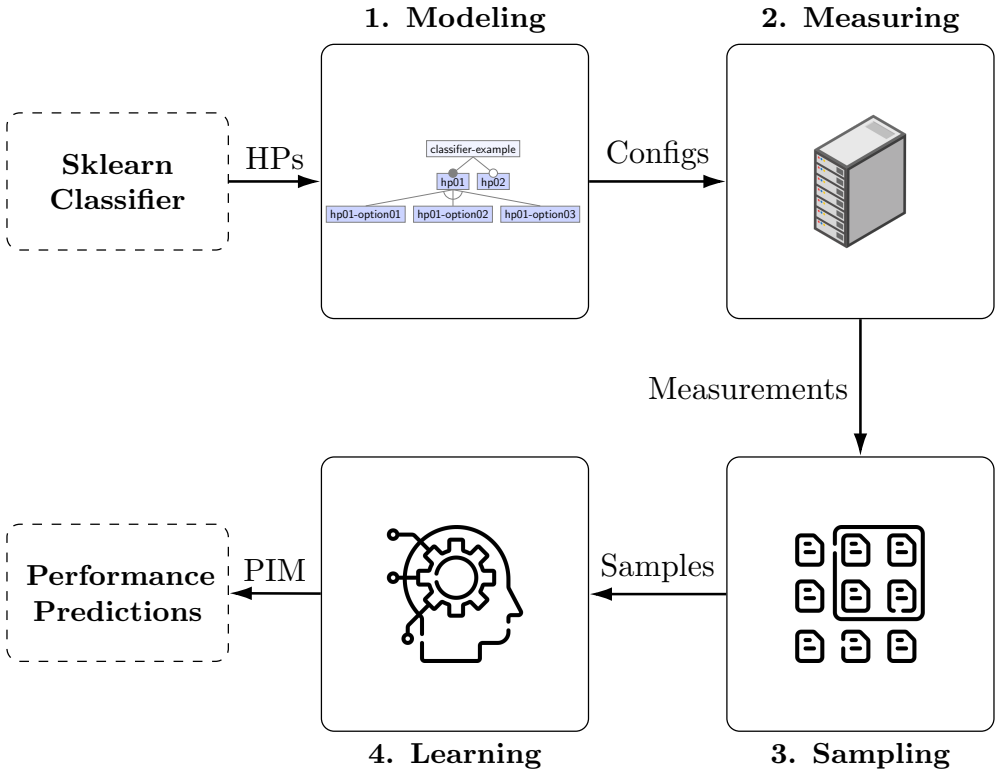
$$\Delta_{\text{acc}} = \max_{c \in \widehat{V}_3} \Big(a_c \; - \; a_{hyperopt} \Big) \tag{4.7}$$

where $\widehat{V}_3$ is the set of 3 configurations with highest $\hat{a}_c$, while a_c is the respective true accuracy. $a_{hyperopt}$ is the best-found accuracy configuration from Hyperopt-sklearn. A positive Δ_{acc} means PIM-guided optimization found a better configuration.

4.3 Evaluation Pipeline

Each experiment will be conducted by running through our evaluation pipeline (see Figure 4.2). We used the idea of the evaluation pipeline from Kaltenecker et al. [11] and adapted it to our needs. The evaluation pipeline consists of four steps, which we will explain in detail in the following sections.

Figure 4.2: Overview of the evaluation pipeline. *HPs* = hyperparameters, *PIM* = performance-influence model.

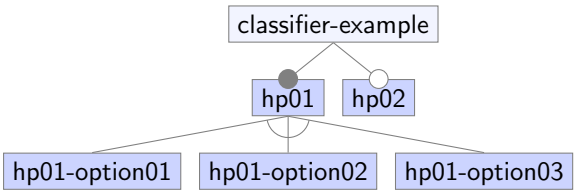


4.3.1 Step 1: Modeling

First, we have to find a way to model an ML problem instance as a configurable software system. This can be achieved by treating the model's hyperparameters as its features that can be tuned. Next, we need to create a FM for each sklearn classifier. By doing so, we define which classifiers to consider and how their hyperparameters and corresponding value space look like. We employ FeatureIDE[1] to create these FMs.

1 https://www.featureide.de/

Example 4. FM representing the structure of the hyperparameter search space for each sklearn classifier. Assume *hpo1* is non-binary and *hpo2* is a binary hyperparameter.



Example 4 is an example of the FM structure that we use for modeling the hyperparameter search space for each classifier. Non-binary features will be subdivided into an alternative-group. Binary features will simply be modeled by an optional feature.

Figure 4.3: Feature Model representing the Gradient Boosting Classifier.

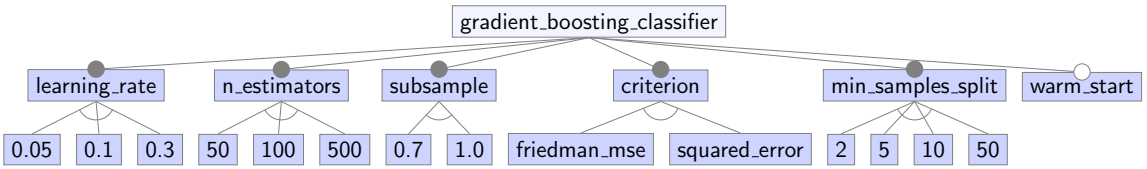


Figure 4.3 showcases our FM of GBC. We limited the search space to six potential sklearn classifiers; the full list of FMs, which we consider, can be found in Appendix A. For each classifier, we further simplified its configuration space where we believed it was appropriate. For this, we use *one-hot encoding*, i.e., we adjusted some non-binary features by turning their continuous value spaces into predefined discrete or categorical options. In Figure 4.3, you can see how feature *n_estimators* has three options (*50*, *100*, *500*), while the full scikit-learn implementation of GBC potentially takes an arbitrary positive integer. To further reduce complexity, we decided to ignore some non-binary hyperparameters entirely (e.g., *min_weight_fraction_leaf*, *max_depth*, or *min_impurity_decrease* for GBC), which means that those parameters will always take the default value.

For the classifiers SGD, KNN, and RFC, there is a hyperparameter *n_jobs* that specifies the number of parallel jobs for both *fit* and *predict*. We set *n_jobs=None*, i.e., no parallelization. The classifiers DTC, GBC, SGD, and RFC also provide a *random_state* hyperparameter to fix the random seed. We set *random_state* to *0*, *1*, *2*, *3* and *4*, in order to run and measure five independent and reproducible trials, and use the average. All other classifiers do not have a random seed option, so we perform only a single run for each.

4.3.2 Step 2: Measuring

Next, we need to automatically collect the ground-truth performance measurements required both for training our PIMs and for assessing their prediction accuracy. For every valid hyperparameter configuration of each classifier, we record its classification accuracy (see Example 5).

Example 5. First ten lines of the final ground-truth performance measurements for GBC. Each line is one hyperparameter configuration, with the last column being the measured accuracy (Equation 4.1). Note that some columns have been omitted for clarity.

```
gbc,criterion,friedman_mse,squared_error,...,warm_start,Performance
1,1,0,1,...,0,0.9637
1,1,0,1,...,1,0.9602
1,1,0,1,...,0,0.9652
1,1,0,1,...,0,0.9461
1,1,0,1,...,1,0.9505
1,1,0,1,...,1,0.8967
1,1,0,1,...,1,0.9459
1,1,0,1,...,0,0.9255
1,1,1,0,...,1,0.8951
...
```

In Table 4.2, we report the mean, median, and standard deviation of the true performance measurements for each classifier which serve as the baseline for all subsequent analyses. Our classifier selection covers varying levels of measured accuracy, with KNN, RFC, GBC, DTC, and MNB achieving high to very high mean accuracies with very low standard deviations, whereas SGD shows a significantly lower mean accuracy and a high standard deviation.

Table 4.2: *Ground-truth accuracy* for each classifier (in %). We provide the *mean* ($\bar{A}$), *median* (med(A)), and *standard deviation* ($\text{œ}(A)$) for each model.

Classifier	$\bar{\mathbf{A}}$	**med(A)**	$\text{œ}(\mathbf{A})$
MNB	83.58	83.60	0.07
DTC	87.78	87.88	0.80
GBC	94.20	94.61	2.17
SGD	62.34	79.54	33.74
KNN	96.83	96.88	0.22
RFC	96.32	96.54	0.63

To streamline and reproduce these measurements, we developed two components:

Main Script Our main script is located in a dedicated repository[2]. It loads the MNIST dataset, iterates over all hyperparameter configurations passed in via command-line arguments, trains the model (here we incorporate sklearn) and outputs the accuracy of each configuration.

VaRA-TS OOT We forked the VaRA-TS out-of-tree framework[3] and added one project per classifier. Each project defines which arguments to use when calling the main script.

2 https://github.com/lukasklein-dev/lk_bachelor_sklearn
3 https://github.com/lukasklein-dev/varats_oot_lukasklein_bachelor

Additionally, we added an experiment class called *SklearnMeasuring*, which defines the steps each project will go through. This also includes building and executing the command for calling the main script.

Cluster To execute the experiments, each classifier project is submitted as a SLURM job on the chair's cluster. For MNB, DTC, SGD, and KNN, we use node *eku* (Intel Core i5-4590 CPU @ 3.30GHz, 16GB RAM, Debian 12). For GBC and RFC, we use node *maxl* (AMD EPYC 72F3 8-Core Processor, 256GB RAM, Debian 12). For MNB and DTC, we set the timeout to 1 hour. For KNN, SGD, and RFC, the timeout was set to 8 hours, while for GBC it was extended to 24 hours. Finally, we merge all performance measurements into a single CSV file per classifier, listing each hyperparameter configuration alongside its measured accuracy.

4.3.3 Step 3: Sampling

Once all ground-truth measurements are in place, we extract a smaller but representative subset of hyperparameter configurations to train our PIMs.

Sampling Techniques We employ two sampling strategies: *diversified distance-based sampling* and *random sampling*, both of which have demonstrated strong performance in prior work [12]. The distance-based approach is particularly well-suited for larger sample sizes, as it aims to maximize diversity without requiring access to the full configuration population. In contrast, random sampling is a straightforward baseline method that also performs well. However, if the sample size is large and the full population is unavailable or sparsely populated with valid configurations, random sampling may frequently select invalid ones, which makes it infeasible in such scenarios. We compare both techniques in our experiments.

Sample Sizes Because the total number of valid configurations varies significantly across classifiers, we fix our sample sizes to *1*%, *5*%, and *10*% of the full configuration set for each classifier (see Table 4.3). These percentages offer a trade-off between training data sufficiency and overall measurement effort, and allow us to compare PIM performance under very low, low, and moderate sample set sizes.

Table 4.3: Overview of the classifiers and their absolute sample sizes.

Classifier	**Sample Size**		
	1%	**5**%	**10**%
MNB	/	1	2
DTC	/	1	2
GBC	3	14	29
SGD	9	45	90
KNN	10	48	96
RFC	13	65	130

4.3.4 Step 4: Learning

Next, we will learn PIMs based on the sample sets to make predictions about the performance of each configuration. In the final step, we will compare our prediction results with the corresponding ground-truth data.

ML Technique We use *stepwise linear regression* to learn a PIM from the sampled performance measurements. Specifically, we fit an *Ordinary Least Squares (OLS)* regression model, treating the hyperparameters as independent variables and the observed performance as the dependent variable. The stepwise procedure iteratively adds or removes hyperparameters based on their statistical significance. This approach allows us to identify the most influential hyperparameters for predicting performance while keeping the model clear and concise.

Implementation We utilize the implementations of the sampling strategies and the regression model on the `ml-sampling` branch of VaRA-feature[4]. To ensure reproducibility, we set `sample_seed = 42`.

Performance Predictions Finally, we map the entire hyperparameter space by using each trained PIM to predict the performance of every configuration, then compare those estimates to the ground-truth measurements from Section 4.3.2 to assess our models' accuracy.

All important data that is related to our experiment, including the FMs, the performance measurements, the learned PIMs, and the evaluation results, will be available in a dedicated repository[5].

4 https://github.com/se-sic/vara-feature
5 https://github.com/lukasklein-dev/lk_bachelor_data

5 Evaluation

In this chapter, we present the results of our experiments and discuss their implications for answering the research questions.

5.1 Results

Following the evaluation pipeline described in Section 4.3, we present the following results: first, the performance-prediction accuracy of our learned PIMs (RQ1); and second, a comparison of our PIM-based hyperparameter optimization against Hyperopt-sklearn (RQ2).

5.1.1 RQ1: Prediction Accuracy

The first research question investigates how accurately PIMs can predict the performance of scikit-learn classifiers across their hyperparameter space (Equation 4.6). The goal is to assess the prediction quality of PIMs under different sampling strategies and sample sizes, and to understand the factors that influence their accuracy.

Following the results presented in Table 5.1, MNB yields nearly identical results under both sampling methods at 5% and 10% sample sizes (MAPE 0.20%), with random sampling achieving a marginal improvement at 10% (MAPE 0.05%). For DTC, random sampling reduces MAPE to 0.73% at 1%, whereas diversified distance-based sampling attains the lowest error of 0.57% at 5%. Both MNB and DTC have a very low spread in their predictions ($\text{œ}(\hat{A}) \leq 0.06\%$ for MNB and $\leq 0.90\%$ for DTC), which is to be expected given the small scale of their experiments (see Table 4.3).

KNN's prediction error steadily declines to 0.16% at 10%, with both sampling approaches performing practically the same. Similarly, RFC demonstrates uniformly low MAPE ($\leq$ 0.17%) across all sample sizes, with random sampling offering a slight edge at larger samples (0.04% vs. 0.13% at 10%). KNN and RFC both achieve high mean predicted accuracies ($\bar{\hat{A}} \approx$ 96.92% for KNN and $\approx$ 96.33% for RFC) with minimal standard deviations ($\text{œ}(\hat{A}) \leq 0.39\%$ for KNN and $\leq 0.62\%$ for RFC), indicating very consistent predictions.

GBC yields stable performance predictions as well: at 1%, random sampling slightly outperforms distance-based (MAPE 2.27% vs. 2.46%), and this gap persists at 5% and 10%, with random sampling holding errors around 2.27%–2.68% compared to 2.46%–5.22% for distance-based sampling. GBC shows a slightly larger spread in its predicted accuracies,

with $\text{œ}(\hat{A})$ up to 4.27% under diversified distance-based sampling and up to 2.50% under random sampling, reflecting increased variability relative to MNB, DTC, KNN, and RFC.

In contrast, SGD shows extreme MAPE values (exceeding 700%) regardless of increased sample sizes, and diversified distance-based sampling performing even worse than random sampling. SGD has both a low mean predicted accuracy ($\bar{\hat{A}}$ ranging from only 51.18% to 92.42% across sample sizes and strategies) and very high variability ($\text{œ}(\hat{A})$ between 23.94% and 87.80%). This mirrors the high variability in its true performance measurements (see Table 4.2).

Some individual predictions for SGD and GBC were so poor that they even fell outside the valid and meaningful accuracy range $[0, 1]$ (e.g., 2.6564 or -1.1853 were among the worst predictions for SGD), which further amplifies the already high standard deviations in their performance predictions.

Table 5.1: *Performance prediction results.* For each learned performance-influence model (under *diversified distance-based sampling*, or *random sampling* and sample sizes corresponding to *1%*, *5%*, or *10%* of each hyperparameter search space V), we report: *mean* ($\bar{\hat{A}}$), *standard deviation* ($\text{œ}(\hat{A})$), and *Mean Absolute Percentage Error* (MAPE) of the predicted accuracy. We mark the best MAPE per classifier green and the worst red. All data is in %, besides the sample size which shows the absolute number of configurations.

Classifier	**Sample Size**	$\bar{\hat{\mathbf{A}}}$		$\text{œ}(\hat{\mathbf{A}})$		**MAPE**	
	#configs	**diversified**	**random**	**diversified**	**random**	**diversified**	**random**
MNB	/	/	/	/	/	/	/
	1	83.41	83.41	0.00	0.00	0.20	0.20
	2	83.41	83.54	0.00	0.06	0.20	0.05
DTC	/	/	/	/	/	/	/
	1	86.58	88.05	0.00	0.00	1.46	0.73
	2	88.13	88.38	0.90	0.33	0.57	0.79
GBC	3	96.30	93.08	0.21	0.88	2.46	2.27
	14	90.51	93.00	4.27	2.50	5.22	2.68
	29	93.37	93.90	4.15	2.09	3.85	2.39
SGD	9	73.29	51.18	23.94	29.90	1067.80	835.66
	45	92.42	64.43	87.80	32.91	1506.75	845.85
	90	62.97	56.37	52.86	49.62	1094.49	712.00
KNN	10	96.92	96.87	0.14	0.39	0.22	0.34
	48	96.87	96.79	0.14	0.14	0.20	0.20
	96	96.84	96.87	0.003	0.10	0.16	0.16
RFC	13	96.23	96.33	0.59	0.58	0.17	0.10
	65	96.26	96.32	0.62	0.62	0.15	0.04
	130	96.30	96.33	0.60	0.62	0.13	0.04

As visualized in the two heatmaps (Figure 5.1 and Figure 5.2), our results imply that the MAPE generally decreases with larger sample sizes (though only slightly) and the two sampling strategies perform similarly, with random sampling retaining a small advantage.

For a table covering all metrics of the performance prediction results, see Table A.1. For a full listing of the best and the worst performing PIMs for each classifier (based on MAPE), see Table A.2 and Table A.3, respectively.

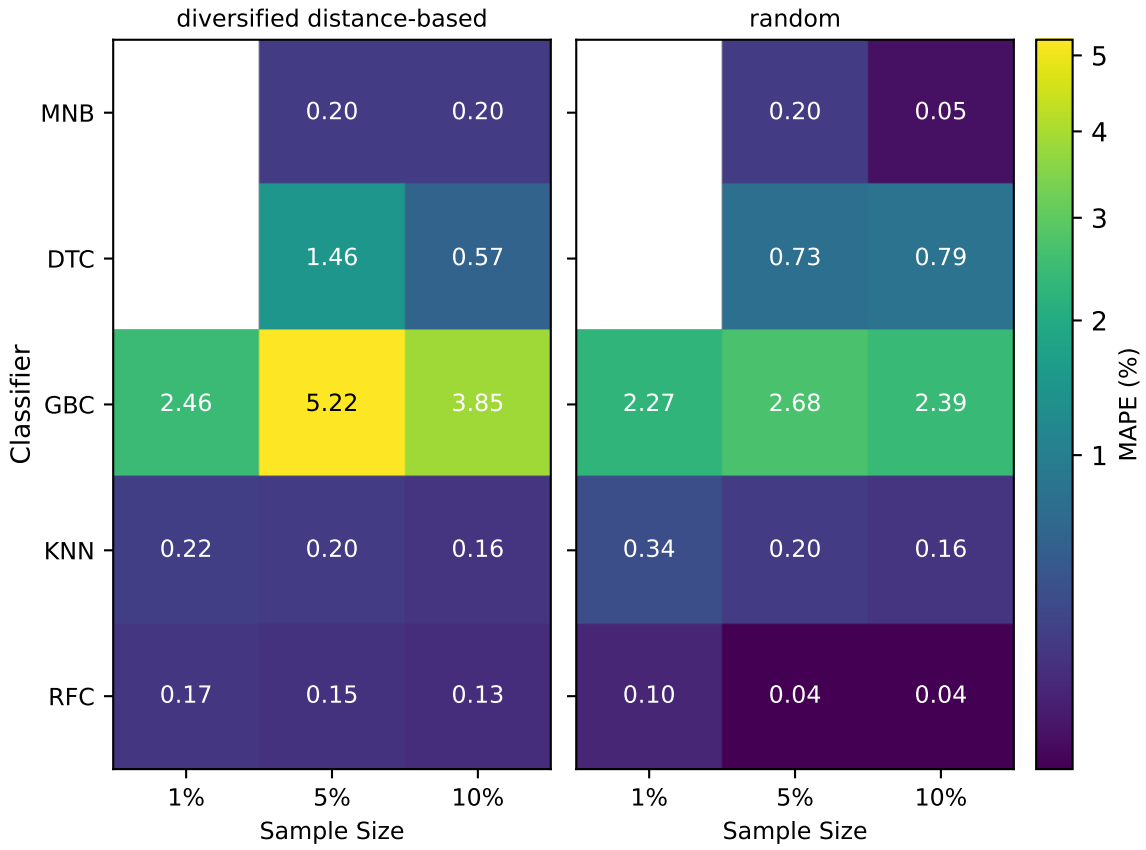


Figure 5.1: Heatmaps of MAPE for every classifier except stochastic gradient descent to prevent scaling issues. Each cell represents one performance-influence model.

> **RQ1 — Results:** Overall, highly accurate performance prediction (MAPE $< 1\%$) is attainable for RFC, MNB, KNN, and DTC using as few as 1%–5% of configurations from the search space, for both sampling strategies. GBC also achieves good prediction accuracy (MAPE 2%–5%) for most sample sizes and random sampling performing slightly better. In contrast, SGD performs the worst with extreme variance in accuracy leading to exorbitant MAPE values.

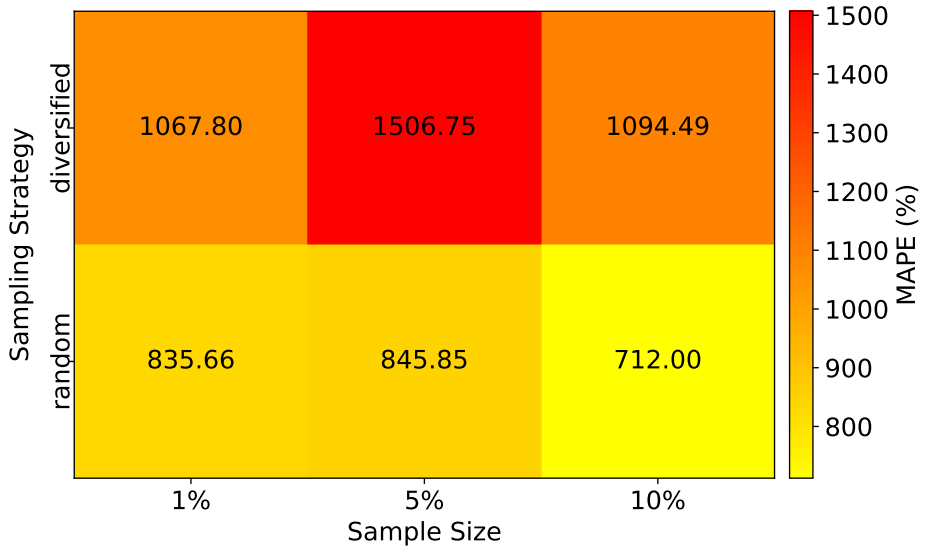


Figure 5.2: Heatmap of MAPE for stochastic gradient descent. Each cell represents one performance-influence model.

5.1.2 RQ2: Comparison to Hyperopt-sklearn

The second research question investigates whether we can use our learned PIMs for efficient hyperparameter optimization, achieving performance comparable to Hyperopt-sklearn under a limited evaluation budget (Equation 4.7).

Table 5.2: *Comparison of the hyperparameter optimization results by our approach and Hyperopt-sklearn.* $\widehat{V}_3$ denotes the top-3 configurations suggested by our approach, V_3 the measured accuracy of the suggested configurations, $a_{hyperopt}$ the accuracy of Hyperopt-sklearn's best configuration, and Δ_{acc} the accuracy difference between the two approaches in percentage points (pp).

Classifier	$\widehat{V}_3$	V_3	$a_{hyperopt}$	Δ_{acc}
MNB	{83.57, 83.57, 83.57}	{**83.64**, 83.64, 83.63}	**83.57**	+0.07
DTC	{88.65, 88.65, 88.65}	{**88.71**, 88.71, 88.65}	**88.27**	+0.44
GBC	{94.33, 94.33, 94.33}	{**97.26**, 97.26, 97.26}	**96.06**	+1.20
SGD	{265.64, 265.64, 256.50}	{79.51, 65.14, **91.43**}	**91.95**	−0.52
KNN	{96.88, 96.88, 96.88}	{**97.17**, 97.17, 97.17}	**97.14**	+0.03
RFC	{96.95, 96.95, 96.95}	{**97.07**, 97.07, 97.07}	**97.01**	+0.06

In Table 5.2, we compare the measured accuracy of the best hyperparameter configuration suggested by our PIM-guided selection to the best one found by Hyperopt-sklearn. For KNN, the top PIM-suggested configuration achieves 97.17% accuracy versus Hyperopt-sklearn's 97.14% (+0.03 pp). RFC attains 97.07% against 97.01% (+0.06 pp). For MNB, the optimal configuration hits 83.64% accuracy, compared to Hyperopt-sklearn's 83.57% (+0.07 pp). DTC attains 88.71% against 88.27% (+0.44 pp). GBC even reaches 97.26% compared to 96.06%,

which results in an improvement of 1.20 pp. However, for SGD our approach suggests three configurations with a predicted accuracy of around 265%, which is not semantically meaningful given that accuracy is defined within the interval [0,1]. The actual accuracy of one such configuration reaches 91.43%, whereas Hyperopt-sklearn's best configuration achieves 91.95%, indicating that our method performs 0.52 pp worse.

> **RQ2 — Results:** Our PIM-guided hyperparameter optimization method suggests configurations that match or slightly outperform Hyperopt-sklearn's optimization results on five of the six classifiers (KNN +0.03 pp; RFC +0.06 pp; MNB +0.07 pp; DTC +0.44 pp; GBC +1.20 pp) and underperform modestly on SGD (−0.52 pp).

5.2 Discussion

After presenting our experiment results, we discuss their possible implications.

5.2.1 RQ1: Prediction Accuracy

The results in Section 5.1.1 demonstrate both the strengths and limitations of our approach to mapping the hyperparameter space. MNB, DTC, KNN, and RFC proved robust, exhibiting narrow accuracy ranges and consistent performance within the defined hyperparameter spaces (see our FMs in Appendix A). In contrast, GBC and especially SGD were highly sensitive, showing large variance and notable mean-median discrepancies indicative of outliers, and consequently higher MAPE. This behavior indicates an insufficient linear relationship between the hyperparameters and the resulting accuracy for SGD and GBC, which likely worsened the effectiveness of OLS linear regression and leads to higher prediction errors (MAPE) for their PIMs.

We therefore hypothesize that the combination of very high standard deviation in performance and a lack of linear dependence between hyperparameters and accuracy is a key factor contributing to the poor predictive performance of the PIMs for SGD. This finding underscores the importance of the modeling step, where domain knowledge can be leveraged to select hyperparameters that are more likely to yield stable outcomes and exclude certain hyperparameter configurations from the beginning that are prone to instability. Indeed, one could argue that classifiers whose performance is highly sensitive to hyperparameter changes are of particular interest, as their complex mapping could yield valuable insights into the hyperparameter space. However, there must be a balance: sufficient variability to reveal meaningful patterns, yet not so much that MAPE becomes unacceptably large.

Originally, we wanted to include four more classifiers: support vector, bayesian gaussian mixture, extra trees, and gaussian process. However, we encountered issues during the measuring step, where they exceeded a runtime limit per configuration; some of them took significantly longer than 24 hours. In addition, while measuring RFC, we encountered a conflict between the *bootstrap* and *oob_score* parameters, which was an oversight in the initial modeling step. To resolve this, we fixed *bootstrap* to *True* for the RFC experiment.

> **RQ1 — Discussion:** Overall, we observe that classifiers highly sensitive to hyperparameter tuning (such as SGD and GBC), and with a weak linear relationship between hyperparameters and accuracy, undermine OLS linear regression and lead to very high MAPE. This highlights the importance to apply domain expertise during the modeling phase to select hyperparameters that achieve a balance between (1) offering enough variability to reveal meaningful patterns and (2) remaining stable enough to keep MAPE low enough, in order to learn PIMs that accurately map the hyperparameter space (as demonstrated by MNB, DTC, KNN, and RFC).

5.2.2 RQ2: Comparison to Hyperopt-sklearn

Our comparison in Section 5.1.2 shows that PIM-guided hyperparameter tuning can match or even slightly exceed the performance of Hyperopt-sklearn. By fitting a PIM to a smaller subset of configurations and then choosing the top three predicted hyperparameter settings, we found true accuracies on par with those returned by the TPE. This confirms that, under tight evaluation constraints, an interpretable PIM can serve not only as a fast global predictor but also as an effective parameter tuner. However, it is crucial to remember that our hyperparameter domains were restricted to the most relevant ranges. Since Hyperopt-sklearn normally explores a wider range of values, we can only make a fair comparison within the smaller parameter ranges we defined. How PIM-guided optimization would behave on a much larger search space remains an open question.

Notably, GBC achieves the largest performance gain over Hyperopt-sklearn, even though it showed the second-highest MAPE in RQ1 (after SGD—which performed the worst in RQ2 as well). We hypothesize that this is because GBC strikes a favorable balance between variation and stability: its performance varies enough to offer substantial improvement potential, yet not so erratically as to destroy the underlying linear relationship between hyperparameters and accuracy. Consequently, even though the PIM for GBC shows a relatively high MAPE, it can still identify high-performing configurations.

When using our learned predictive models for hyperparameter optimization, minimizing MAPE is less important than preserving the correct ordering of configurations by performance. In practice, as long as the model ranks configurations in the same relative order as their true accuracies, we can identify the top-performing hyperparameter setting without requiring highly precise point estimates. Consequently, even models with modest overall predictive accuracy can drive effective optimization (as seen for GBC), provided they maintain a consistent monotonic relationship between predicted and actual performance.

Beyond hyperparameter optimization, the real advantage of our approach lies in the global performance map it produces. Rather than returning just one "best" configuration, the learned PIM covers the expected accuracy across the entire hyperparameter landscape. This allows us to pinpoint which areas of the hyperparameter space consistently deliver high or low performance, and to explore parameter interactions to identify exactly which combinations improve accuracy or worsen it. In essence, PIM-guided tuning delivers both a competitive optimizer and a rich, interpretable model of hyperparameter-accuracy relationships, making it a powerful tool for efficient and insightful hyperparameter analysis.

> **RQ2 — Discussion:** Our PIM-guided optimization performed comparably to Hyperopt-sklearn's results within our constrained parameter ranges. In this context, maintaining the correct ranking of configurations is even more important than producing highly precise point estimates, allowing models with higher MAPE to still identify top performers. GBC's notable gain demonstrates that a hyperparameter space with sufficient variability yet enough stability can yield substantial improvements.

5.3 Threats to Validity

In this section, we discuss the threats to internal and external validity, to ensure the reliability and applicability of our findings.

5.3.1 Internal Validity

One key threat to internal validity is our discretization of continuous hyperparameter spaces: by restricting values (for example, limiting the number of trees or learning rates to a smaller selection of discrete options), we may miss important regions or optima. To mitigate this, we selected value options based on scikit-learn's own defaults and our domain knowledge, which typically perform reasonably well out of the box.

Reproducibility of our performance measurements is another concern: small differences in randomness or execution environment can lead to divergent results. We addressed this by fixing all random seeds, averaging each classifier's accuracy over five independent runs, and running every experiment under the same cluster environment.

Finally, our use of ordinary least squares regression assumes linearity and constant variance of residuals, assumptions that break down for highly non-linear learners. Indeed, for SGD and GBC some PIM predictions fell outside the valid $[0, 1]$ accuracy range, producing nonsensical values. While this highlights a limitation of unbounded linear models, we note that future work could employ clipping or a logit-transform plus sigmoid back-transform to enforce valid bounds and stabilize variance near the extremes.

5.3.2 External Validity

Our study concentrates on six widely used scikit-learn classifiers (Multinomial Naive Bayes, Decision Trees, Gradient Boosting, Stochastic Gradient Descent, K-Nearest Neighbors, and Random Forest). This selection ensures in-depth analysis of their hyperparameter-accuracy relationships but necessarily restricts the scope of our conclusions. Other classifiers (e.g., Support Vector Machines, or Bayesian Gaussian Mixture) often have fundamentally different hyperparameter spaces and interactions. As a result, the PIM behaviors and sampling sensitivities we observe here may not hold for models whose performance is strongly non-linear. Extending our framework to such algorithms will likely require revisiting both the choice of sampling budgets and the form of the regression model.

Similarly, we only evaluated two sampling strategies, namely uniform random and diversified distance-based sampling, which means our findings on MAPE may not hold for alternative methods (e.g., t-wise sampling). As a result, any application of our PIM framework to other classifiers or sampling paradigms should be preceded by a fresh validation.

6 Concluding Remarks

6.1 Conclusion

In this thesis, we investigated whether sampling-based performance-influence models can accurately predict the performance of ML hyperparameter configurations and serve as a competitive alternative to Bayesian optimization techniques such as those used by Hyperopt-sklearn. To this end, we answered two research questions:

Research Question 1

Summary By modeling each of six scikit-learn classifiers as a configurable system and applying both diversified distance-based and random sampling at budgets of 1%, 5%, and 10% of the full configuration space, we trained PIMs via stepwise ordinary least squares (OLS) regression. For classifiers with relatively smooth, low-variance accuracy landscapes (Multinomial Naive Bayes, Decision Tree, K-Nearest Neighbors, Random Forest), our PIMs achieved exceptionally low MAPE ($\leq$ 1% at just $1\% - 5\%$ sample size). For Gradient Boosting, prediction error remained modest ($\approx 2 - 5\%$ MAPE), while Stochastic Gradient Descent, characterized by highly variable and non-linear performance, proved unsuitable for a PIM, resulting in unacceptably high error. These results demonstrate that, when the hyperparameter-accuracy relationship is sufficiently linear and stable, sampling-based PIMs can model performance accurately.

Research Question 2

Summary Using the top-3 configurations predicted by each classifier's best PIM and comparing against Hyperopt-sklearn's best configuration, we found that PIM-guided optimization matched or slightly exceeded Hyperopt-sklearn on five of six classifiers (gains of $+0.03$ pp to $+1.20$ pp in accuracy). Only for Stochastic Gradient Descent, where the PIM broke down, did Hyperopt-sklearn perform better (-0.52 pp). Thus, within controlled hyperparameter ranges, interpretable PIMs can deliver competitive tuning performance.

Beyond matching Hyperopt-sklearn, our approach provides a global performance map of the entire configuration space, highlighting not just a single optimum but the full landscape of how each hyperparameter and interaction affects accuracy.

6.2 Future Work

Although our study has shown that PIMs trained via stepwise linear regression on diversified distance-based and random sampling budgets can accurately model classifier performance and guide hyperparameter selection, several important extensions remain.

So far, we have ignored the runtime and efficiency dimension; future work should also consider the time required to collect performance measurements and train PIMs, in order to evaluate whether our approach is not only effective in terms of accuracy but also efficient in practice. This includes exploring how to reduce the time needed for performance measurement, which can be significant for complex classifiers, large datasets, and unconstrained hyperparameter search spaces.

Consequently, another limitation of our approach arises: we constrained our evaluation to six scikit-learn classifiers and further simplified each configuration space by hard-coding default hyperparameter settings where deemed appropriate and by transforming continuous features into predefined discrete options. While these choices reduced overall complexity, they may also have introduced bias and limited the generalizability of our findings. To address this, subsequent research should relax these simplifications, explore richer, unconstrained hyperparameter spaces without manual defaults or coarse discretization, and assess how PIMs perform under higher-dimensional, more heterogeneous domains. This also includes running experiments on GPUs, since during the collection of performance measurements for several classifiers we observed some configurations exceeding a timeout of 24 hours. Thus, reducing the runtime of the measuring step can enable the exploration of more complex PIMs and therefore larger hyperparameter spaces.

Appendix

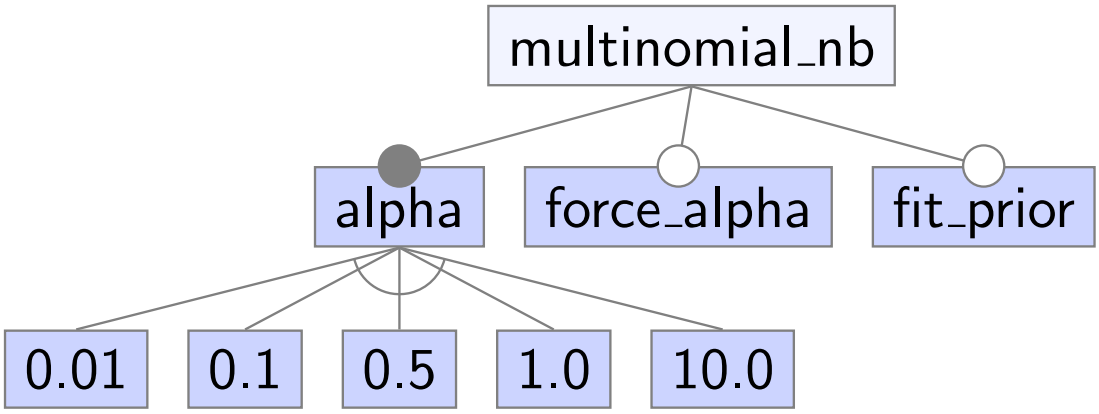


Figure A.1: Feature Model representing the Multinomial Naive Bayes Classifier.

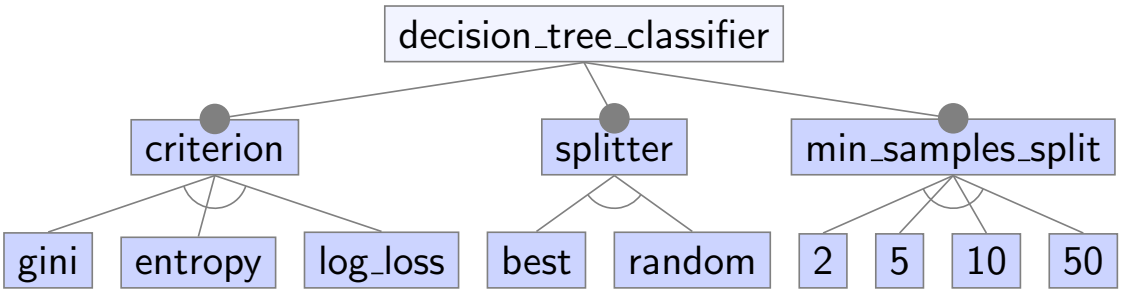


Figure A.2: Feature Model representing the Decision Tree Classifier.

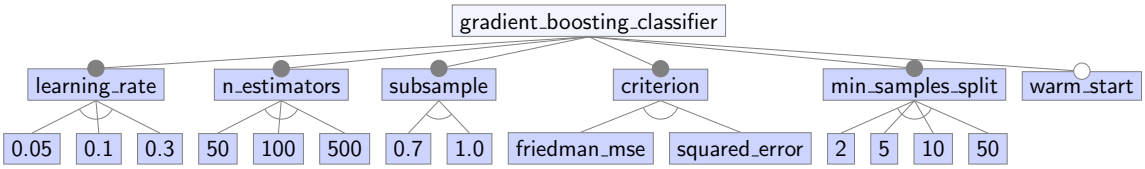


Figure A.3: Feature Model representing the Gradient Boosting Classifier.

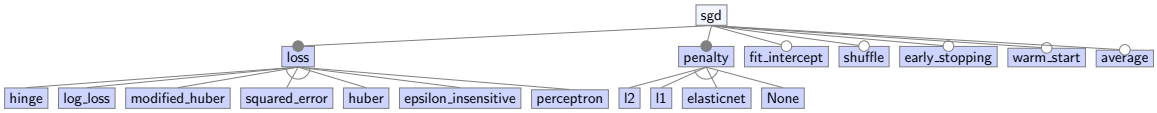


Figure A.4: Feature Model representing the Stochastic Gradient Descent Classifier.

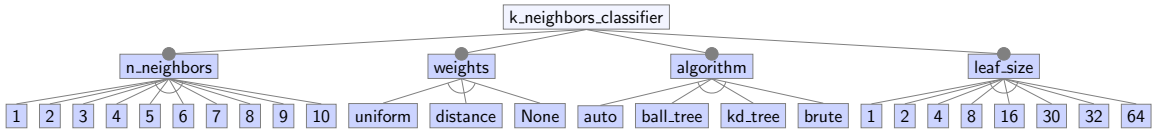


Figure A.5: Feature Model representing the K-Nearest Neighbors Classifier.

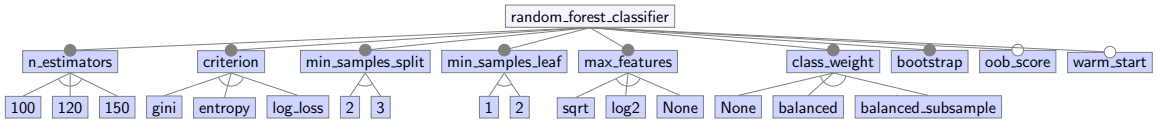


Figure A.6: Feature Model representing the Random Forest Classifier.

Table A.1: Predicted performance by classifier, sampling strategy and sample size. All metrics are in % and are further defined in Section 4.2.2. $\Delta_{\text{Mean}} = \bar{\hat{A}} - \bar{A}$, and $\Delta_{\text{Median}} = \text{med}(\hat{A}) - \text{med}(A)$. A positive Δ would indicate an over-estimation, while a negative Δ indicates an under-estimation. The 95%-Bootstrap CI-L and CI-U columns report the lower and upper bounds of a 95% confidence interval, estimated via bootstrap resampling.

Classifier	Sampling Strategy	Sample Size	$\bar{\hat{A}}$	$\text{med}(\hat{A})$	$\text{œ}^2(\hat{A})$	$\text{œ}(\hat{A})$	Δ_{Mean}	Δ_{Median}	CI-L	CI-U	MAPE
MNB	diversified-distance	/	/	/	/	/	/	/	/	/	/
		1	83.41	83.41	0.00	0.00	−0.17	−0.19	83.41	83.41	0.20
		2	83.41	83.41	0.00	0.00	−0.17	−0.19	83.41	83.41	0.20
	random	/	/	/	/	/	/	/	/	/	/
		1	83.41	83.41	0.00	0.00	−0.17	−0.19	83.41	83.41	0.20
		2	83.54	83.57	0.00	0.06	−0.04	−0.03	83.51	83.56	0.05
DTC	diversified-distance	/	/	/	/	/	/	/	/	/	/
		1	86.58	86.58	0.00	0.00	−1.20	−1.30	86.58	86.58	1.46
		2	88.13	88.65	0.80	0.90	0.35	0.77	87.79	88.48	0.57
	random	/	/	/	/	/	/	/	/	/	/
		1	88.05	88.05	0.00	0.00	0.27	0.17	88.05	88.05	0.73
		2	88.38	88.38	0.11	0.33	0.60	0.50	88.24	88.52	0.79
GBC	diversified-distance	3	96.30	96.37	0.04	0.21	2.10	1.76	96.28	96.33	2.46
		14	90.51	89.51	18.22	4.27	−3.69	−5.10	90.02	91.00	5.22
		29	93.37	94.56	17.22	4.15	−0.83	−0.05	92.87	93.85	3.85
	random	3	93.08	92.46	0.78	0.88	−1.12	−2.15	92.99	93.19	2.27
		14	93.00	94.07	6.26	2.50	−1.20	−0.54	92.70	93.29	2.68
		29	93.90	93.02	4.35	2.09	−0.30	−1.59	93.65	94.14	2.39
SGD	diversified-distance	9	73.29	86.16	572.86	23.94	10.95	6.62	71.72	74.81	1067.80
		45	92.42	90.59	7708.51	87.80	30.08	11.05	86.77	98.22	1506.75
		90	62.97	66.78	2793.70	52.86	0.63	−12.76	59.44	66.38	1094.49
	random	9	51.18	57.11	893.80	29.90	−11.16	−22.43	49.30	53.10	835.66
		45	64.43	72.62	1083.36	32.91	2.09	−6.92	62.33	66.55	845.85
		90	56.37	59.88	2462.17	49.62	−5.97	−19.66	53.23	59.58	712.00
KNN	diversified-distance	10	96.92	96.92	0.02	0.14	0.09	0.04	96.91	96.93	0.22
		48	96.87	96.91	0.02	0.14	0.04	0.03	96.86	96.88	0.20
		96	96.84	96.88	0.00	0.003	0.01	0.00	96.84	96.85	0.16
	random	10	96.87	96.88	0.15	0.39	0.04	0.00	96.85	96.90	0.34
		48	96.79	96.87	0.02	0.14	−0.04	−0.01	96.78	96.80	0.20
		96	96.87	96.91	0.01	0.10	0.04	0.03	96.86	96.87	0.16
RFC	diversified-distance	13	96.23	96.45	0.35	0.59	−0.09	−0.09	96.20	96.27	0.17
		65	96.26	96.48	0.38	0.62	−0.06	−0.06	96.29	96.33	0.15
		130	96.30	96.52	0.36	0.60	−0.02	−0.02	96.26	96.33	0.13
	random	13	96.33	96.55	0.34	0.58	0.01	0.01	96.30	96.36	0.10
		65	96.32	96.54	0.38	0.62	0.00	0.00	96.29	96.36	0.04
		130	96.33	96.54	0.38	0.62	0.01	0.00	96.29	96.36	0.04

Table A.2: Overview of *the best performance-influence model per classifier* (i.e., the one with smallest MAPE). We report mean ($\bar{\hat{A}}$), median ($med(\hat{A})$), standard deviation ($\sigma(\hat{A})$), and Mean Absolute Percentage Error (MAPE) of the predicted accuracy. All data is in %, besides the sample size which shows the absolute number of configurations.

Classifier	Sampling Strategy	Sample Size	$\bar{\hat{\mathbf{A}}}$	$\mathbf{med}(\hat{\mathbf{A}})$	$\mathbf{\sigma}(\hat{\mathbf{A}})$	MAPE
MNB	random	10%	83.54	83.57	0.06	0.05
DTC	diversified-distance	10%	88.13	88.65	0.90	0.57
GBC	random	1%	93.08	92.46	0.88	2.27
SGD	random	10%	56.37	59.88	49.62	712.00
KNN	diversified-distance	10%	96.84	96.88	0.003	0.16
RFC	random	5%	96.32	96.54	0.62	0.04

Table A.3: Overview of *the worst performance-influence model per classifier* (i.e., the one with largest MAPE). We report mean ($\bar{\hat{A}}$), median ($med(\hat{A})$), standard deviation ($\sigma(\hat{A})$), and Mean Absolute Percentage Error (MAPE) of the predicted accuracy. All data is in %, besides the sample size which shows the absolute number of configurations.

Classifier	Sampling Strategy	Sample Size	$\bar{\hat{\mathbf{A}}}$	$\mathbf{med}(\hat{\mathbf{A}})$	$\mathbf{\sigma}(\hat{\mathbf{A}})$	MAPE
MNB	random	5%	83.41	83.41	0	0.20
DTC	diversified-distance	5%	86.58	86.58	0	1.46
GBC	diversified-distance	5%	90.51	89.51	4.27	5.22
SGD	diversified-distance	5%	92.42	90.59	87.80	1506.75
KNN	random	1%	96.87	96.88	0.39	0.34
RFC	diversified-distance	1%	96.23	96.45	0.59	0.17

Statement on the Usage of Generative Digital Assistants

For this thesis, the following generative digital assistants have been used: We have used ChatGPT-4o (OpenAI, version March 2025) [18] for ideation, rephrasing, and structuring ideas. We are aware of the potential dangers of using these tools and have used them sensibly with caution and with critical thinking.

Bibliography

[1] Justus A Ilemobayo, Olamide Durodola, Oreoluwa Alade, Opeyemi J Awotunde, Adewumi T Olanrewaju, Olumide Falana, Adedolapo Ogungbire, Abraham Osinuga, Dabira Ogunbiyi, Ark Ifeanyi, Ikenna E Odezuligbo, and Oluwagbotemi E Edu. "Hyperparameter Tuning in Machine Learning: A Comprehensive Review." In: *Journal of Engineering Research and Reports* 26.6 (2024), 388–395. DOI: 10.9734/jerr/2024/v26i61188. URL: https://journaljerr.com/index.php/JERR/article/view/1188.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[3] J. Bergstra, D. Yamins, and D. D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures." In: *TProc. of the 30th International Conference on Machine Learning (ICML 2013), June 2013, pp. I-115 to I-23.* (2013).

[4] Yi-Wei Chen, Qingquan Song, and Xia Hu. "Techniques for Automated Machine Learning." In: *SIGKDD Explor. Newsl.* 22.2 (Jan. 2021), 35–50. ISSN: 1931-0145. DOI: 10.1145/3447556.3447567. URL: https://doi.org/10.1145/3447556.3447567.

[5] Li Deng. "The MNIST database of handwritten digit images for machine learning research." In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[6] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. "A genetic algorithm for optimized feature selection with resource constraints in software product lines." In: *Journal of Systems and Software* 84 (Dec. 2011), pp. 2208–2221. DOI: 10.1016/j.jss.2011.06.026.

[7] Huong Ha and Hongyu Zhang. "DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 1095–1106. DOI: 10.1109/ICSE.2019.00113.

[8] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. "Combining multi-objective search and constraint solving for configuring large software product lines." In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, 517–528. ISBN: 9781479919345.

[9] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated Machine Learning: Methods, Systems, Challenges - Chapter 5*. Springer, 2019.

[10] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. "An algorithm for generating t-wise covering arrays from large feature models." In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. SPLC '12. Salvador, Brazil: Association for Computing Machinery, 2012, 46–55. ISBN: 9781450310949. DOI: 10.1145/2362536.2362547. URL: https://doi.org/10.1145/2362536.2362547.

[11] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "The Interplay of Sampling and Machine Learning for Software Performance Prediction." In: *IEEE Software* 37.4 (2020), pp. 58–66. DOI: 10.1109/MS.2020.2987024.

[12] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. "Distance-Based Sampling of Software Configuration Spaces." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 1084–1094. DOI: 10.1109/ICSE.2019.00112.

[13] Shubhra Kanti Karmaker Santu, Md. Mahadi Hassan, Micah Smith, Lei Xu, Chengxiang Zhai, and Kalyan Veeramachaneni. "AutoML to Date and Beyond: Challenges and Opportunities." In: *ACM Comput. Surv.* 54.8 (Oct. 2021). ISSN: 0360-0300. DOI: 10.1145/3470918. URL: https://doi.org/10.1145/3470918.

[14] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. "Fast Bayesian hyperparameter optimization on large datasets." In: *Electronic Journal of Statistics* 11 (2017). URL: https://api.semanticscholar.org/CorpusID:4938627.

[15] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. "Tradeoffs in modeling performance of highly configurable software systems." In: *Softw. Syst. Model.* 18.3 (June 2019), 2265–2283. ISSN: 1619-1366. DOI: 10.1007/s10270-018-0662-9. URL: https://doi.org/10.1007/s10270-018-0662-9.

[16] Brent Komer, James Bergstra, and Chris Eliasmith. "Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn." In: *Proceedings of the 13th Python in Science Conference*. Ed. by Stéfan van der Walt and James Bergstra. 2014, pp. 32 –37. DOI: 10.25080/Majora-14bd3278-006.

[17] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. "Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features." In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. SPLC '19. Paris, France: Association for Computing Machinery, 2019, 289–301. ISBN: 9781450371384. DOI: 10.1145/3336294.3336297. URL: https://doi.org/10.1145/3336294.3336297.

[18] OpenAI. *ChatGPT-4o: Language Model*. 2025. URL: https://chat.openai.com.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[20] Atrisha Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. "Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T)." In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 342–352. DOI: 10.1109/ASE.2015.45.

[21] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. "Taking the Human Out of the Loop: A Review of Bayesian Optimization." In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

[22] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-influence models for highly configurable systems." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, 284–294. ISBN: 9781450336758. DOI: 10.1145/2786805.2786845. URL: https://doi.org/10.1145/2786805.2786845.

[23] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. "Predicting performance via automated feature-interaction detection." In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 167–177. DOI: 10.1109/ICSE.2012.6227196.

[24] Xilu Wang, Yaochu Jin, Sebastian Schmitt, and Markus Olhofer. *Recent Advances in Bayesian Optimization*. 2022. arXiv: 2206.03301 [cs.LG]. URL: https://arxiv.org/abs/2206.03301.

[25] Jia Wu, Xiu-Yun Chen, Hao Zhang, Li-Dong Xiong, Hang Lei, and Si-Hao Deng. "Hyperparameter Optimization for Machine Learning Models Based on Bayesian Optimization." In: *Journal of Electronic Science and Technology* 17.1 (2019), pp. 26–40. ISSN: 1674-862X. DOI: https://doi.org/10.11989/JEST.1674-862X.80904120. URL: https://www.sciencedirect.com/science/article/pii/S1674862X19300047.