INSTITUTE FOR PARALLEL AND DISTRIBUTED SYSTEMS

SIMULATION TECHNOLOGY DEGREE COURSE

Bachelor thesis

Submitted to the University of Stuttgart

# Comparison of different Hyperparameter-Tuners for Support Vector Machines

An analysis using Parallel Least-Squares SVM Library on GPU

Examiner

Prof. Dr. Dirk PFLÜGER

Institute for Parallel and Distributed Systems

First Supervisor

M.Sc. Peter Domanski

Institute for Parallel and Distributed Systems

Second Supervisor

M.Sc. Alexander Van Craen

Institute for Parallel and Distributed Systems

Third Supervisor

M.Sc. Marcel Breyer

Institute for Parallel and Distributed Systems

Submitted by

| | |
|---|---|
| Author | Yannick Marian DZUBBA |
| Matriculation number | 3401428 |
| SimTech-Nr. | 184 |
| Submission date | April 2024 |

## Abstract

Working with large datasets requires sophisticated tools. One such tool developed for classification is the Support Vector Machine (SVM). As with any ML algorithm, the user has to set several different Hyper Parameter (HP) to run a SVM. Finding the optimal choice of HPs is important for model performance and it is highly dependent on the dataset. Given the number of different HPs, a search space might be massive, so optimization methods have been developed, to automate this search. This work aims to compare three popular choices: The Grid Search, the Random Search and Bayesian Model Search. They are compared in different metrics, such as performance, runtime and energy. Optuna [ASY+19] was used as optimizer backend, it implements all three optimizer types, it implements Tree-Parzan Estimator (TPE) as Bayesian Search algorithm. It was connected to Parallel Least-Squares Support Vector Machine (PLSSVM) [VCBP22] as SVM implementation. PLSSVM can efficiently exploit parallel compute cores. The optimizers have been tested on a selection of different search spaces and datasets with PLSSVM running on Graphic Processing Unit (GPU).

# Contents

# List of Figures

# List of Tables

# Acronyms

**C** cost. 61

**CG** conjugate gradient. 26

**CNN** Convolutional Neural Network. 78

**CPU** Central Processing Unit. 26

**CSV** Comma-Separated Values. 31

**CV** k-fold cross-validation. 7

**d** polynomial degree. 64

**GPU** Graphic Processing Unit. 3

**HP** Hyper Parameter. 3

**HPO** Hyper Parameter Optimization. 7

**IPVS** Institute for Parallel and Distributed Systems. 13

**k** kernel. 59

**ML** Machine Learning. 3, 13

**NN** Neural Network. 78

**oaa** one against all. 28

**oao** one against one. 28

**OS** Operating System. 55

**PLSSVM** Parallel Least-Squares Support Vector Machine. 3

**poly** polynomial. 28

**PSV** proportion of support vectors. 29

**RAM** Random-Access Memory. 40

**rbf** Radial Basis Function. 28

**sc** scaling HP. 59

**SMO** Sequential Minimal Optimization. 26

**SVM** Support Vector Machine. 3

**TDP** Thermal Design Power. 55

**TPE** Tree-Parzan Estimator. 3

**VRAM** Video Random-Access-Memory. 35

# 1 Introduction

To make predictions based on data, a widely used algorithm is the Support Vector Machine. As with any Machine Learning (ML) algorithm, with a SVM the user is faced with the question, of what parameters to use. The correct choice of these parameters is important so that algorithm can find a good model for the data. These parameters are referred to as HP.

Given this premise, the question arises, how can the user find HP, that will result in a good model? In the case of a SVM, there are a number of different HP to consider, such as the *kernel*, the *cost*, or the *tolerance*. Depending on the *kernel*, there are even more HP (for example the *degree* parameter of the *polynomial* kernel function). Due to the number of HPs, a *search space* (subsection 2.4.3) might be massive, and it is not easy to guess which combination of HPs will lead to a good model. To compound this fact, fitting any ML model usually requires significant resources, both in terms time and energy. Given these constraints, several automatic optimization methods have been developed, to help find a good model.

Three popular methods are the *Grid Search*, the *Random Search* and the search guided by a *Bayesian Model* over the HP. In this work, the Optuna [ASY+19] package was chosen, it implements all three of the above optimizers. TPE was used as the Bayesian Model Search. For the SVM implementation, the PLSSVM library [VCBP22], developed by Van Craen et al. at the Institute for Parallel and Distributed Systems (IPVS) was chosen. PLSSVM takes advantage of the massive amount of compute cores found on a modern GPU to parallelize the work of fitting the model using the Least-Squares approach, thereby reducing the required runtime. This is particular important because a reduction in runtime allows using a bigger search space to be used for´ optimization.

To compare these three HP optimizers, several tests were developed, each increasing the search space covered by the optimizers. The tests were run on three different datasets, that may be classified using the PLSSVM library. These tests aim to find out which optimizer works well for a given search space and dataset. All three optimizers are compared in terms of model performance, optimization runtime and energy consumption. Additional tests were developed, to find approaches, how to deal with search spaces, where a full search might exhaust the available resources. The development of the tests is explained in chapter 3.

This task was accomplished by implementing a comprehensive HPO framework, which combines Optuna and PLSSVM with components for loading datasets and search spaces [Dzu24] and then performing the optimization studies. A high-level overview of the framework is given in section 3.3.

# 2 Propraedeuticum

This chapter discusses works related to our research. It includes the works on which this research was based and works that aim to research the same field of HPO for SVM.

## 2.1 Metrics

When evaluating any model, there are two broad categories to consider. First, a trained model will have some performance on a task. For example, a metric is used to quantify how well a model trained on handwritten numbers can predict the correct digits. On the other hand, training such a model requires resources in terms of time and energy. The latter two are logged during model fitting, which is described in subsection 3.3.5. Grandini et al. wrote a comprehensive paper on the topic of multiclass classification metrics [GBV20], which was used as basis for this section.

### 2.1.1 Model performance metrics

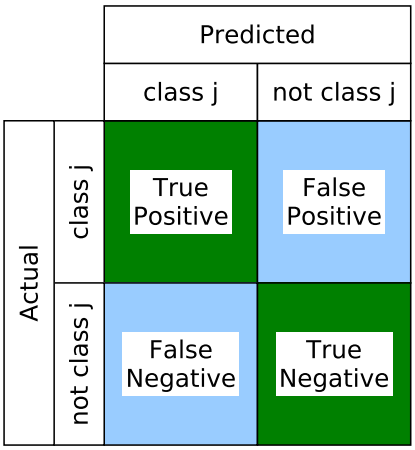



**Figure 2.1:** A confusion matrix for a single label.

Since the scope of this work is limited to classification, only classification metrics are discussed. A classifier is a supervised ML function that, given an input, predicts which class the data belongs to. To evaluate the performance of a classifier, there are a handful of different metrics that can be used.

These metrics exist per class and aggregated for all classes. The performance of a single class $c_j$ within the classifier can be evaluated using the following set of metric equations:

$$p_j = \frac{TP_j}{TP_j + FP_j} \tag{2.1}$$

$$r_j = \frac{TP_j}{TP_j + FN_j} \tag{2.2}$$

$$f1_j = \frac{2 p_j r_j}{p_j + r_j} \tag{2.3}$$

$TP_j$ denotes the true positives of class j. Or in other words, how often the classifier correctly identified data belongs to class j. This is usually visualized with the aid of a confusion matrix, as in Figure 2.1. $p_j$ denotes precision and $r_j$ the recall of class j. The $f1_j$ is the harmonic mean of recall and precision.

$$F1 = \frac{1}{m+1} \sum_{j=0}^{m} f1_j \tag{2.4}$$

$$TP = \sum_{j=0}^{m} TP_j \tag{2.5}$$

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.6}$$

$$E = 1 - Acc \tag{2.7}$$

$Acc$ is the *accuracy* score of the model. Accuracy is a very commonly reported metric and has been used to compare tuning results with each other in section 4.3 and with literature section 4.5. In some cases, the *misclassification error* $E$ is reported, which is closely related to accuracy. It has the advantage that when plotted on a logarithmic scale, the difference between better performing models may be more visible. An example of such a plot can be seen in Figure 3.8. The $F1$ denotes the macro score, which is the unweighted average of all $f1_j$ scores. Accuracy and weighted metrics have the disadvantage that, in the case of an unbalanced dataset, they may still report a good aggregated score even if the performance of a less represented class is bad [GBV20, cpt. 4.2]. Conversely, we noticed that the $F1$ score was sometimes undefined. This behavior is due to it's definition and it depends on the use case if such a behavior is desirable or not. For all these reasons, both the accuracy and the $F1$ metrics are implemented in the PLSSVM-HPO framework developed (subsection 3.3.4). All defined metrics are in the range $[0, 1]$. For all of them, higher is better, except for the misclassification error.

## 2.2 Dataset

A classification dataset is defined as the the data $X$ and the corresponding labels $Y$. Some practical considerations on how to prepare a dataset are given in section 3.1. Without loss of generality, each data point of $\vec{x_i}$ can be written as a one-dimensional vector $\begin{bmatrix} x_{i,0} & x_{i,1} & \dots & x_{i,m} \end{bmatrix}$. The formal definition of a dataset $D$ is

$$X = \begin{bmatrix} x_{0,0} & x_{0,1} & \dots & x_{0,m} \\ x_{1,0} & x_{1,1} & \dots & x_{1,m} \\ & & \dots & \\ x_{n,0} & x_{n,1} & \dots & x_{n,m} \end{bmatrix}, \quad Y = \begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_n \end{bmatrix} \tag{2.8}$$

$$D = [X, Y]\,. \tag{2.9}$$

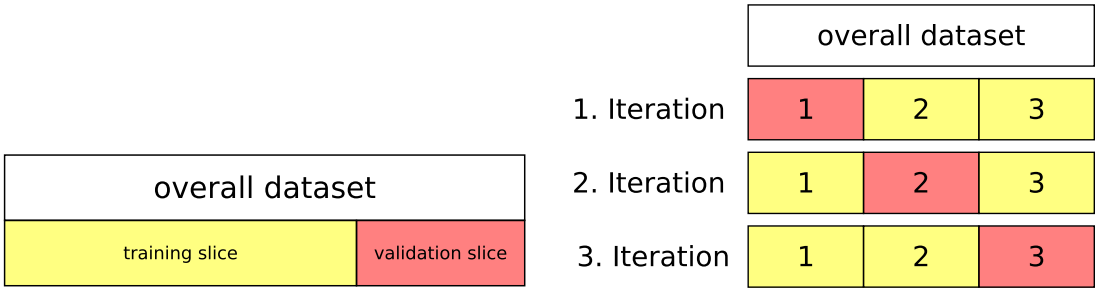
Each label entry $y_i$ is part of a set of unique class labels $L$. Usually, the size of $L$ is much smaller, then the size of $Y$. $L$ consists of either discrete or categorical data. Splitting the dataset is crucial for assessing a model's generalization. For this purpose, define a sample as $d_i = (\vec{x_i}, y_i)$. Splitting a dataset can be done by defining a new dataset over indices $J$. This approach was adapted from [BBL+21, cpt. 3.2].

$$D_J = \begin{bmatrix} d_{j_0} \\ \dots \\ d_{j_k} \end{bmatrix}, \quad \forall j_i \in J \subset [0, n] \tag{2.10}$$

Notice, that $n$ is defined as the numbers of samples in the original dataset $D$ Equation 2.9. The model trains on a part of the data $D_{train}$. After training, the model may have a good classification performance on the training dataset. In case of accuracy, this would be called the *training-accuracy*. However, a high training accuracy only provides insight into how the model can classify previously seen data. For this reason, a high training accuracy does not imply a good overall model. It is not uncommon for a model with good training accuracy to perform poorly on unseen data.It is not uncommon, that a model with a good training-accuracy performs poorly on unseen data. This is a well-known phenomenon known as overfitting. Practical examples of overfitting in SVM can be seen in [Tha19]. To get a better idea of the generalization performance of a model, it is used to predict unseen data. Two common *model evaluation* approaches are the *holdout strategy* and *CV*.

### 2.2.1 Holdout strategy

With holdout, a slice of the dataset, a part is split off the dataset as validation dataset, while the remaining slice is used for training. This gives two dataset slices, one $D_{train}$ and one $D_{validation}$. The model is trained on $D_{train}$ and assessed on $D_{validation}$, which is used for model selection. It is sketched in Figure 2.2a. For datasets with few samples, this may lead to a case, where the small validation dataset is not representative of the entire dataset. Bischl et al. call this a pessimistic bias. In this case, they recommend using CV.




**(a)** Visualization of the holdout strategy.

**(b)** Visualization of 3-fold cross validation, the yellow slices are the training slice and the red slices are used for validation of each iteration.

**Figure 2.2:** Comparison of model evaluation strategies.

### 2.2.2 K-fold cross-validation

In contrast to the holdout strategy, in CV not one, but k models are trained. The dataset is split in $k$ parts, $D_1$ to $D_k$. Using the same HPs, k different models are trained. Each time one of the slices is used evaluate the performance of that model, while the other $k-1$ slices are used for training. For example, in the first iteration, the model is trained on slices 2 to $k$ and evaluated on slice 1. In the second iteration, the model trains on slices 1 and 3 up to $k$ and it is evaluated on slice 2. This process is repeated $k$-times, an example of a 3-fold cross validation is sketched in Figure 2.3. To get a final result, the result of each model is aggregated, often using the arithmetic mean. Of the $k$ different models, one is then randomly chosen at random, and the other $k-1$ models are discarded. While this approach can reduce the pessimistic bias, it has the disadvantage, that training $k$ different models takes about $k$ times the resources.

Given that the training time is usually dependent on the size of the dataset, this is a valid trade-off. A dataset with few samples, that may benefit from CV will usually be fit in less time. As a result, the computational overhead of CV may still be feasible. On the other hand, fitting a large data set $k$ times may not be feasible. According to the research of Bischl et al., the pessimistic bias of a larger dataset should be smaller and therefore CV may not be needed. Because of this tradeoff, both strategies are implemented in the PLSSVM-HPO framework (subsection 3.3.5).

## 2.3 Hyper Parameter

In general, HPs can be divided into two broad categories: model HPs and non-model HPs. The category of model HPs includes all HPs that are directly part of the model algorithm directly. This would be the *kernel* or the *cost* parameters in the case of a SVM.

This set is strictly limited by the used ML algorithm implementation used. Non-model HP may include parameters, such as the feature scaling subsection 3.1.4 used for the dataset. Since these parameters do not depend on the algorithm itself, there is no clear demarcation of what may or may not be considered as a HP. A practical definition might be that if a user tunes a certain parameter to optimize a model, then that parameter can be called to as a Hyper Parameter.

In this work, a combination of HPs is referred to as $\vec{\lambda}$, while a single HP of this combination is written as $\lambda_k$. HPs can be of any data type, whether numeric (continuous or discrete), categorical, or boolean. A single HP $\lambda_k$ is usually a scalar value. Any HP can be defined for certain ranges. For example, the polynomial degree HP of a SVM is a strictly positive integer.

## 2.4 Hyper Parameter Optimization

There are many HPO frameworks available, such as Optuna [ASY+19], Ray Tune [LLN+18] or frameworks built into general purpose ML libraries, such as GridSearchCV in Scikit-learn [PVG+12]. This section is based on Bischl et al., which gives a praxis oriented overview of the concept of HPO.

For any HPO framework, the user specifies the search space per HP and some stop criteria. Usually, the user can also choose which type of sampler to use. The repeated process of sampling HP and evaluating them on the *objective function* $\omega$ (subsection 2.4.2) is called a *study*. Each study runs, until at least one stop criterion (subsection 2.4.5) is met. At the end, the best model according to $\omega$ is selected. The best model is validated by measuring it's predictive performance on the *holdout split* (section 2.2). Given this performance, the user can choose to use the model, or to reject it and rerun the study.

### 2.4.1 HPO validation strategy

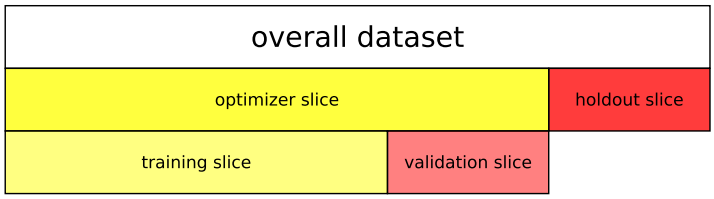



**Figure 2.3:** Visualization of all dataset slices of a simple HPO. In this example, the dataset is split into three parts: One to train the model, one to validate the trained model and one to assess the performance of the tuned model. CV can be done on the optimizer slice as well.

Evaluating the final performance of the HPO can run into the same problem, as evaluating a model section 2.2. Namely, optimizing the HPs optimizes the final model for samples of the dataset, which can lead to overfitting. In the case of the holdout strategy, this can lead to a model overfit for the dataset part $D_{validation}$. For this reason, the dataset is split into two overarching pieces: The $D_{optimizer}$ optimizer dataset and the $D_{holdout}$ validation holdout dataset. The optimizer trains and evaluates the model on $D_{optimizer}$ which itself is split once again. When the optimizer finds the best model, it's performance is evaluated again, but this time on $D_{holdout}$. This performance should be less biased, because the model never had contact with this data. To differentiate the strategies, this approach might be referred to as *holdout-validation*. Another approach might be, to use the same HPs as used by the best model and retrain them on $D_{holdout}$. A name for this approach could be *holdout-training*. The idea behind holdout training is that if the optimizer has found a *stable*

*optimum*, then a model trained on comparable data should have a performance near that optimum as well. A small deviation in the input data of $\omega$ (in this case the dataset) should still yield comparable performance. Whereas if the optimizer has found a *sharp optimum*, then the small change in the input data results in much worse performance. While both strategies can be used, the holdout training strategy requires $D_{holdout}$ to be of comparable size of $D_{optimizer}$ for it to be comparable. This may reduce the size of the latter and thus the performance of the final model. Therefore, the strategy depends on the datasets to be evaluated. In our case, the objective of the PLSSVM-HPO framework is to find a well-performing model, not necessarily to find HPs that are likely to result in a good model. Any model, that performs well on holdout validation would suffice.

### 2.4.2 Objective function

Let $f(\vec{\lambda}, D_i)$ be an arbitrary classification function. $\vec{\lambda}$ are the HP input arguments of the function, $D$ is the dataset used. The classification function $f$ is trained on the training slice, but the performance metric can be applied to any of the dataset slices, which is denoted by the $D_i$. To evaluate the fitness of $f$, a metric $\mu$ is applied. An overview of different metrics is given in section 2.1. This leads to the *objective function*

$$\omega(\vec{\lambda}) = \mu(f(\vec{\lambda}, D_i)). \tag{2.11}$$

In this work, the function $f$ would fit a model and then evaluate it using PLSSVM. This process is also referred to as a *trial*.

### 2.4.3 Search space

Any optimization problem begins with the definition of the search space. For each Hyper Parameter (HP) $\lambda_k$, a search space $\Lambda_k$ is defined. The type of space of $\Lambda_k$ depends on the HP and the sampler to be used. It can be *categorical*, *discrete* or *continuous*. In cases, where $\lambda_k$ is defined for a continuous space, $\Lambda_k$ can be defined to be continuous or discrete. A sampler-specific restriction may be, that $\Lambda_k$ may only consist of discrete or categorical data in the case of a Grid Search. $\Lambda_{degree}$ = [2, 3, 4, 5] is an example of a search space of the *degree* HP of a SVM with a *polynomial* kernel. For continuous or discrete definition spaces of $\lambda_k$, the sampler usually requires $\Lambda_k$ to have a lower bound and an upper bound. A general search space is constructed as:

$$\Lambda = \Lambda_1 \times \Lambda_2 \times ... \times \Lambda_d, d \geqslant 1 \tag{2.12}$$

To denote the a search space $\Lambda_k$ of a single HP from the combined search space $\Lambda$, the single search space $\Lambda_k$ is referred to as the $k$-th *axis* of the search space $\Lambda$. Given this definition, even a relatively small dimension $d$ can result in massive search spaces, due to the *curse of dimensionality*. The dimension of $\Lambda$ may be smaller than the number of HP of the function $f$. In such a case, default values are used for the HP not present in the search space. Finding a search space $\Lambda$ is discussed in section 3.4.

### 2.4.4 Sampler

In any HPO framework, the sampler is responsible for picking a combination of HP from the search space $\Lambda$. It builds up the HP combination $\vec{\lambda}$ from the values $\lambda_k$ sampled from each search axis $\Lambda_k$ and the default values of the function $f$, for each HP. These are then used to evaluate the objective function $\omega$. W.l.o.g., this leads to the primary optimization function of

$$\vec{\lambda} = argmax\ \omega(\vec{\lambda}), \quad \vec{\lambda} \epsilon \Lambda. \tag{2.13}$$

Given the fact, that for most of the defined metrics higher is better (subsection 2.1.1), the optimization function maximizes the objective function. Sampling of some search spaces $\Lambda_k$ is conditional, on other search spaces $\Lambda_g$, or default values. In the case of the SVM, the degree HP is used only for the polynomial kernel. Thus, a sampler may only draw different values for degree, if the kernel was determined to be polynomial. In this example, sampling the kernel search space must take precedence, if it is contained in the overall search space $\Lambda$. *Conditional sampling* has to take default values into account. If the kernel is set to be polynomial by default, then the degree search space must be sampled, even if the kernel is not part of the search space. Figure 2.6 shows a sketch of the behavior of the grid-search, the random-search and the Bayesian Model Search.

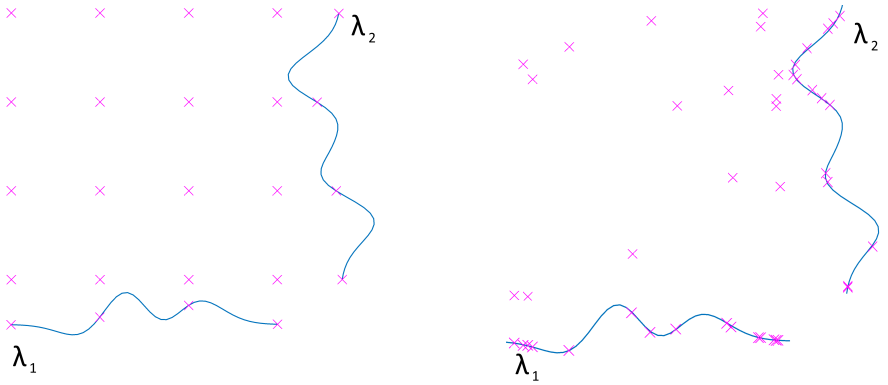
#### Grid Search

As indicated to in subsection 2.4.3, a Grid Search requires a discrete search space. For a continuous space, this means that some specific points must be selected. The user can either specify each value. Alternatively, they can specify the limits and the scale. In the case of a linear scale, they can discretize using the number of steps in the interval, or the length of each step. For the logarithmic scale, the step size is not defined and the number of steps has to be used for discretization. Typically, a Grid Search samples a random combination, that has not been evaluated yet. Although some Grid Search may use a simpler for-each loop approach, that samples the values in a determined order. In this work, a Grid Search was used that randomly samples combinations. A Grid Search may have the theoretical advantage, of covering the entire search space, if the search is not interrupted by some other stopping criterion. However, this may not hold true for large search spaces, where a complete Grid Search may exhaust the available resources. Another disadvantage may be, that a grid may miss any optimum that lies outside the discretization.

#### Random Search

For a Random Search any bounded search space can be used. While the basic idea is similar to the Grid Search, the Random Search does not require a discretization. HP on continuous spaces are drawn randomly. This has the theoretical advantage, that the user does not have to consider how to discretize the data, but only the scale of the data. Another advantage may be, that it evaluates combinations that would fall outside any discretization. A disadvantage may be, that it cannot guarantee, that the search space was searched has been searched evenly. Note that in Figure 2.4a, all corners are searched and the space is searched more evenly than in Figure 2.4b. On the other hand, Figure 2.4b shows that more samples are tried per axis.

Assume a two-dimensional search space $\Lambda$ and Grid Search over four by four combinations (as in Figure 2.4a). Now if another continuous search axis was added $\Lambda^* = \Lambda \times \Lambda_3$, which is to be evaluated on four points as well, then the grid of $\Lambda^*$ will contain 64 combinations. But the original search space $\Lambda$ is still evaluated on the 16 points, as at the beginning of the thought experiment. While using Random Search, the original search space $\Lambda$ is likely to result in 64 evaluated points. Bergstra and Bengio argue that if $\Lambda_3$ is less important for the objective function, then introducing the new axis will significantly reduce the performance of the Grid Search, as will require four times the function evaluation for almost no difference in the result. Whereas it will have little effect impact on Random Search. Bergstra and Bengio describe this as, the Random Search has a *lower effective dimensionality* compared to the Grid Search.




**(a)** Grid Search **(b)** Random Search

**Figure 2.4:** Recreation of figure in [BB12]. Sketch of a two-dimensional, continuous search space. In both examples, exactly 16 samples were drawn. The position of each sample is drawn once as combination and once on each search axis. This example tries to optimize the function $f(\vec{\lambda}) = a(\lambda_1) + b(\lambda_2)$, $\vec{\lambda} = [\lambda_1, \lambda_2]$.

### Bayesian Model Search

Another popular class of samplers are Bayesian Model Search. The same user considerations apply to Bayesian Model Search as to Random Search. In each trial, the score is stored in an *archive*. To build up the archive, a Bayesian Model Search starts by executing a set number of trials of Random Search. After that, the sampler constructs a *surrogate model* using the archive $A$. One part of the surrogate $\hat{\omega}(\vec{\lambda})$ maps the HP to the expected value of the objective function, which is usually modeled using a Gaussian process. While another part keeps track of the uncertainty of the HP $\hat{\sigma}(\vec{\lambda})$. The uncertainty of $\hat{\sigma}(\vec{\lambda_i}) = 0, \quad \forall \vec{\lambda_i} \epsilon A$. It is bigger in regions, where the next $\vec{\lambda_i}$ are further apart. Therefore a bigger uncertainty can be thought of as an unexplored region of the objective function. With these two parts, the trade-off between exploration and exploitation can be adjusted. The loop of optimizing the surrogate, suggesting and evaluating samples is sketched in Figure 2.5.

While this surrogate model may still be complex, it's evaluation is cheap. Therefore the surrogate model is optimized to find the next candidate of HPs to try on the expensive objective function. This candidate $\vec{\lambda_i}$ is then evaluated on the objective function, and the result is fed back into the model. The goal of this approach is, for the Model Based sampler to exploit an optimum when

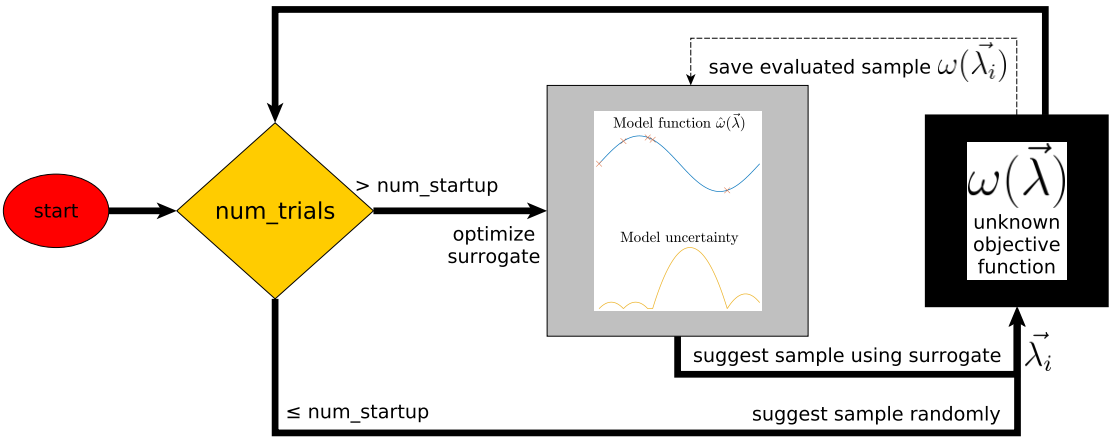
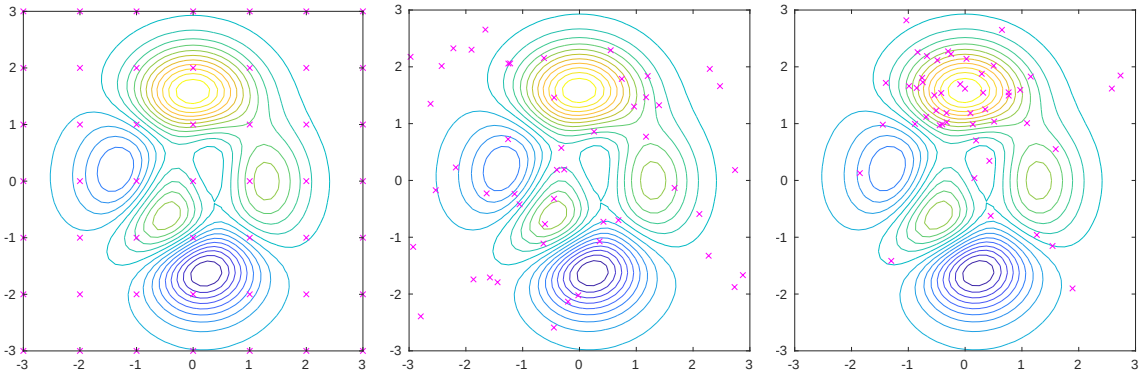



**Figure 2.5:** Sketch of a Bayesian Model Search.

it is found. Due to the model uncertainty, the sampler tries to explore the function, to reduce the risk, of getting stuck in a local optimum. It should also increase the chance, of finding the global optimum. As in Random Search, omitting the grid helps to reduce the effective dimensionality. Unlike in Random Search, adding another axis to an existing search space can make the surrogate model more complex and it may reduce it's performance.

Basic implementations of a Bayesian model search, such as described in [JSW98]) only work with real and unconditional HP search spaces. To allow other data than real numbers to be used in a Gaussian Process, the values of the HP in the model are rounded to the nearest discrete value [GMHL20]. Conditional search spaces can be implemented by using a tree structure, as was done in the TPE sampler [Wat23]. Here the conditional hierarchy is implemented as a tree, with the conditional HPs lower in the tree's structure. Each HP is then sampled, by recursing over each node of the tree. For this reason, the TPE sampler was chosen as the Bayesian Model Search.



**(a)** Grid Search **(b)** Random Search **(c)** Bayesian Model Search

**Figure 2.6:** Sketch of how the sampling of HP on this objective function might look like. The contour plot shows the objective function, while the scatter plot shows the sampled HPs.

#### Evolution Strategy

Bischl et al. identify another popular sampling strategy, the Evolution Search. This strategy has not been compared in this work. The name of this class of algorithms is due to the fact, that they are inspired by the biological concept of evolution. A detailed introduction to this class of algorithms is given by Beyer and Schwefel [BS02]. In this context, each HP configuration $\vec{\lambda}_i$ is called an *individual*. At each iteration of the algorithm, a new *population* of individuals is generated and evaluated. Random Search is used, to generate the first population. The *fitness* of each individual is evaluated using the objective function. For each subsequent iteration, a new population is generated, by randomly mixing the configuration $\vec{\lambda}_i$ of the best individuals, this may be referred to as *reproduction*. Next, the population is expanded, by applying *mutation*. In this step, individuals are copied and in the copies, one or more HPs are randomly changed in the copies. Then the fitness of each individual in the new population is evaluated. The process of reproduction, mutation and evaluation is repeated, until the study is stopped.

One limitation using this approach is, that each individual requires to have the same a combination of the same HPs. If individual A has the sampled HPs {kernel=polynomial, degree=3, cost=1, $\epsilon = 10^{-6}$} and B has {kernel=linear, cost=10, $\epsilon = 10^{-8}$}, then these individuals cannot be mixed. This can be rectified, by always sampling the conditional HPs of the search space, even if they remain unused. For this reason, this approach is not as well suited to conditional search spaces (subsection 2.4.3), as other approaches. Also, for the selection of parents to work effectively, the population of each iteration must be sufficiently big. This makes this type of algorithm more suitable for less expensive objective functions.

### 2.4.5 Stop criteria

In general, two different types of stop criteria can be defined for HPO. One is when to stop a running study, often referred to as *pruning*. This can be done, by fitting multiple models simultaneously. During fitting, the performance of all models are evaluated at regular intervals and the worse performing models are stopped. Another type of stop criteria is when to stop the HPO study. In the context of a PLSSVM, there is no insight into the model while it is fit to the data. Thus, pruning any model was not considered. For the same reason, any model optimization strategy, that uses pruning (e.g. *successive halving* described in [FH19, cpt. 1.4.2]) cannot be used with PLSSVM.

A variety of different stop criteria can be applied to a study, depending on the constraints at hand. One of the simplest stop criterion is to execute a fixed number of trials $n_{trials}$ per study. For random or Bayesian model-based searches, the search space $\Lambda$ is infinite, if at least one of the dimensions is continuous. As such, $n_{trials}$ is an arbitrary stop criterion for these samplers, while it can be defined as the number of unique configurations in the case of a Grid Search. Another class of stop criteria are resource based. A user can set a runtime or energy budget per study. When the budget is exhausted, the study is stopped and the best model found so far is evaluated. A final class of stop criteria is the *score-based* stop. Bischl et al. also identify a simple threshold based approach [BBL+21, cpt. 6.5], if the objective function $\omega$ exceeds a set threshold $\tau$, then the study is stopped and this model is validated. This can be a good approach, if the user has a target performance in mind and has a rough idea of what performance to expect from a given dataset and objective function. They also point out two possible drawbacks of such an approach. If set too low, the optimizer may find a suboptimal model. On the other hand, if it is set too high, and the optimizer

may never reach the threshold. Therefore, for this particular approach, a secondary constraint, such as a runtime limit or number of trials must be set. Another score-based criterion is formulated in [RDPCV+18]. If the top 25% of models have a score variance less than 0.01, then the study is stopped. This is likely to work with a Bayesian Model Search, or an Evolution Strategy. However, there is a risk that a search algorithm like Grid Search won't converge, because the results of Grid Search have a higher variance, compared to other optimizers (as can be seen in section 4.3). A simpler form of such a convergence criterion was developed as part of this work (section 4.4).

## 2.5 Support Vector Machine

Developed by Cortes and Vapnik, the Support Vector Machine is one of many tools used for classification problems. This chapter aims to present the core ideas of [CV95].

### 2.5.1 Basic working principle

Recall the definition of the dataset section 2.2. In the case of a basic SVM, all values are $x_{i,j} \epsilon \mathbb{R}$, and without loss of generality, the labels are defined as $y_i \epsilon \{-1, 1\}$. Any SVM implemented using this simple scheme is restricted to a one-dimensional vector with m entries and binary classification. An example of a simple SVM is sketched in Figure 2.7. Given these constraints, the SVM determines the m-dimensional hyperplane, that can separate the data with the greatest margin. This plane is also known as the *decision boundary*. It is defined as $\langle \vec{w}_i, \vec{x}_i \rangle + b = 0$. $\vec{w}_i$ defines a vector orthogonal to the decision boundary, $b$ describes the distance to the origin. This leads to the primary optimization inequality

$$y_i(\langle \vec{w}_i, \vec{x}_i \rangle + b) \geqslant 1 \quad \forall i. \tag{2.14}$$

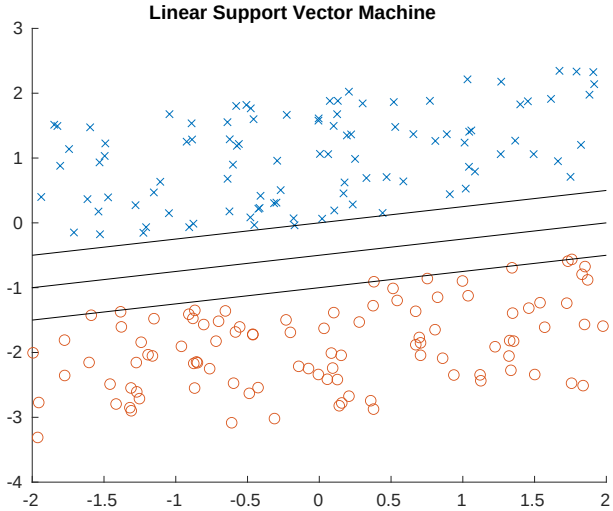



**Figure 2.7:** Simple example of a SVM on two-dimensional data

### 2.5.2 Soft margin hyperplane

The SVM model constructed so far works for cases, where the data is separable using a hyperplane. This model may only work if the margin of the decision boundary is positive. If this is not the case and the data is not linearly separable, the model will never find any decision boundary. The soft margin hyperplane solves this problem by reformulating the objective. Instead of just maximizing the decision boundary margin, missclassification is allowed. Now, the boundary margin is maximized, while the margin violation error is also minimized. This is achieved by introducing a slack variable $\vec{\xi}$. The rate, at which this error minimization is done is controlled by the cost Hyper Parameter $C$.

$$min\left(\frac{1}{2}||\vec{w}||_2^2 + C\sum_i \xi_i\right), \quad C > 0, \quad \forall i, \tag{2.15}$$

defines the boundary violation error (using the form defined in [VCBP22]), while the the main inequality is changed to

$$y_i(\langle \vec{w}_i, \vec{x}_i \rangle + b) \geqslant 1 - \xi_i, \quad \xi_i \geqslant 0, \quad \forall i. \tag{2.16}$$

### 2.5.3 Parallel Least-Squares Support Vector Machine

To implement the SVM optimization problem, many libraries, such as the popular LIBSVM library [CL11], use an algorithm known as Sequential Minimal Optimization (SMO) [Pla98]. While SMO is a proven algorithm, Van Craen et al. identify its sequential nature as a bottleneck for parallel computing. As a result, these types of algorithms have difficulty efficiently exploiting multicore hardware. blem is reformulated as an equality [SV99]:

$$y_i(\langle \vec{w}_i, \vec{x}_i \rangle + b) = 1 - \xi_i, \quad \forall i. \tag{2.17}$$

Now the problem can be solved using the *Least-Squares method*, for which massively parallel algorithms are well known. In the implementation of PLSSVM the *conjugate gradient (CG)* method [HS52] was used to solve the Least-Squares problem [VCBP22]. Van Craen et al.'s primary motivation was to develop a Least-Squares SVM library, that works with systems that use different software and hardware components. It can efficiently utilize the many cores of modern Central Processing Units (CPUs) and GPUs, thereby significantly reducing the runtime of fitting each SVM model. To solve the matrix, PLSSVM implements two modes, the *explicit* and the *implicit* CG. In the explicit mode, the matrix is completely assembled and then solved. In implicit mode, parts of the matrix are calculated, as soon as they are used in the calculation. As a result, in explicit mode, the algorithm uses more memory and the initial setup of the matrix takes more time. Whereas in implicit mode, each iteration takes more time to compute.

### 2.5.4 Kernel

The approach described so far has the drawback, that it only works for data, that can be separated by a hyperplane. The data visualized in the example in Figure 2.8, cannot be separated using a hyperplane with any reasonable performance.

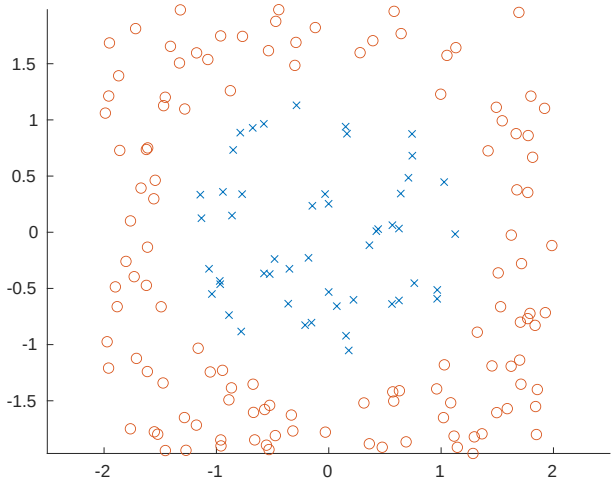



**Figure 2.8:** This example is not linearly separable. Any linear decision boundary, would result in a model with bad classification performance.

This problem is solved by introducing the kernel function. The kernel function is applied to the data $X$ to transform it to a higher dimension. Instead of separating the data directly, the SVM now separates the transformed data. This means, the kernel function is an important Hyper Parameter to consider. See Figure 2.9 for an example of this approach.

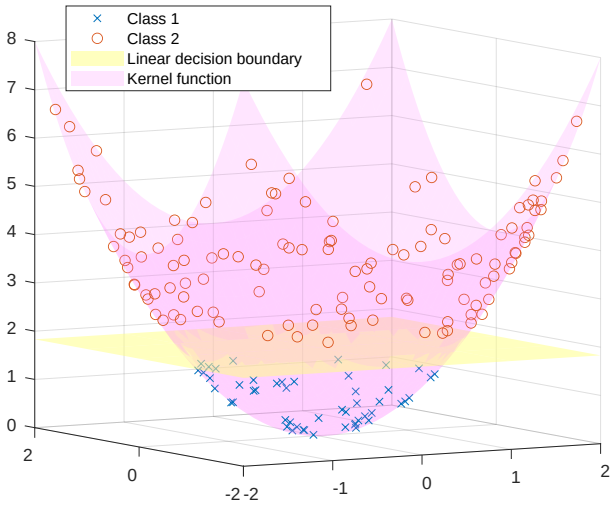



**Figure 2.9:** To solve this problem, the data is transformed from a two-dimensional space to a three-dimensional space using a second-degree polynomial kernel. Now a linear decision boundary can separate the two classes again.

(2.18)
$$linear\colon\ \langle x, x' \rangle$$
(2.19)
$$poly\colon\ (\gamma \langle x, x' \rangle + c_0)^d$$
(2.20)
$$rbf\colon\ exp(-\gamma ||x - x'||^2).$$

PLSSVM supports the *linear*, *polynomial (poly)* and *Radial Basis Function (rbf)* kernels. The short forms of poly and rbf were adopted from SVM interface libraries, such as Scikit-Learn [PVG+12].

### 2.5.5 Multiclass classification

Currently, the SVM model is restricted to data with labels $\{-1, 1\}$. To get around this limitation, data with more than two unique labels are split up into multiple binary models. Two popular approaches are the one against one (oao) approach (also known as one versus one) and the one against all (oaa) (also known as one versus all) [MG13]. The classification labels are defined by the following equation. $L$ is the set of labels of the dataset, $L$ has $k$ different unique entries,

(2.21) $y_i \epsilon \{l_0, l_1, ..., l_k\} = L,\ k > 2.$

In oao classification, one model is constructed for each non-trivial, unordered combination of $L \times L$. This means that only one of the models $\{l_1, l_0\}$ and $\{l_1, l_0\}$, only one is used. Trivial models of the type $\{l_j, l_j\}$ are omitted. As a result, one binary model is constructed for each possible label combination, resulting in $\frac{k(k-1)}{2}$ unique binary models. In PLSSVM, one Least-Squares problem is solved for each model.

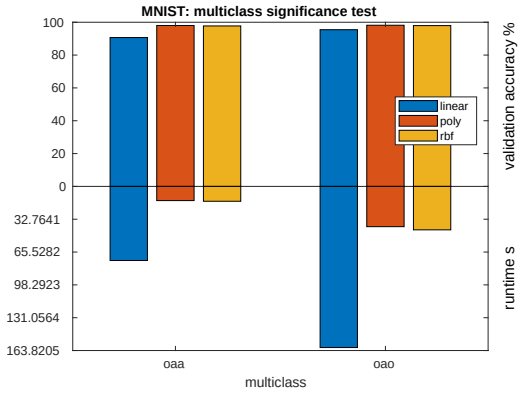



**Figure 2.10:** Multiclass classification model performance and runtime. Uses the same HPs described in the significance tests in section 3.4. Accuracy measured on validation slice defined in section 2.2.

Another common approach is oaa, where one binary model is constructed for each label. Model $j$ classifies, whether a sample belongs to class to class $j$, or to any other class. This is formalized as the model $\{l_j, \neg l_j\}$, where $\neg l_j$ is defined for all $L_j = L \backslash \{l_j\}$. In PLSSVM, this is implemented, by using a single CG matrix that is solved for each label.

Due to the difference in how each approach classifies data, both approaches can lead to models with different performance. For example, when using a linear kernel, oao tends to increase the model performance, which is shown in Figure 2.10. This is probably due to the fact, that combining several classes $\neg j$ in oaa makes them less linearly separable from class $j$. This means that the type of classification is an important HP for multiclass classification problems[1].

### 2.5.6 Support Vector Regression

To allow the SVM to be used for regression, Scholkopf et al. proposed a modification of the established SVM algorithm. Instead of maximizing the decision boundary, the SVM regression algorithm aims to minimize the boundary around the data. This allows the SVM to predict continuous values. Tuning of a SVM regression model is beyond the scope of this work. Optimization for SVM regression follows the same principles, but it would use differnet metrics, to measure model performance.

## 2.6 Related works HPO analysis for SVM

This section lists some of the related research on the topic of HPO with SVM. [MRV+15] explore an extensive set of search algorithms and datasets, to optimize the $\gamma$ and the cost parameter. As search algorithms, they compare particle swarm optimization, model-based search, Random Search, Grid Search and genetic algorithm, which includes the three algorithms of our research. Their list of datasets includes some popular examples, such as the Iris dataset [Fis36], or the wine dataset [AF91]. All their datasets are simple, only five out of 70 datasets have more than 50 features, which is not comparable to our dataset selection section 2.2. Their research itself is mainly concerned with the raw model performance. While their title suggests that they mainly studied Random Search, their findings seem to indicate, that the choice of search algorithms depends on the dataset.

The paper [RDPCV+18] also includes an extensive selection of datasets. As was the case in [MRV+15], the selection uses only simple datasets with only few samples and features. They compared Bayesian Model Search with several evolutionary algorithms. With *proportion of support vectors (PSV)*, they optimize for an interesting secondary objective.

$$PSV = \frac{num_{support\_vectors}}{num_{training\_samples}} \tag{2.22}$$

can be used as an indicator of generalization. The lower the metric, the simpler the model, and therefor the lower the risk of an overfit model. They emphasize, that a more complex model is more likely to be overfit. Furthermore, they formulate a stop criterion, that stops the study early when convergence is observed. In our research, we formulate a comparable, but simpler criterion in section 4.4. While they use PSV to formulate a *efficiency*, they did not include any information about runtime versus performance.

[1] At the time the experiments ran, the oao could not be used. This should be addressed in future research.

[CSB15] studies the use of the same classes of search algorithms as our research on bioactive compounds datasets. Their results suggest, that in most cases, the Bayesian search is the best choice for their application in terms of performance. While also mainly concerned with performance, some plots of iteration and accuracy are included. With these plots, they show, that Grid Search takes longer than any other search to find the best solution. This may not be the best way, to represent model performance. At least in our experience, the runtime of each SVM trial may vary somewhat. For this reason, a plot of score against runtime may be a better idea. The plot is further limited by the fact, that it does not convey any information about the variance over multiple trials of the same search.

Compared to these two, [KFB+17] contains extensive graphs of runtime against error. They have developed an interesting Bayesian-based algorithm called *Fabolas*. It adapts the size of the dataset, to find a tradeoff between computational cost and information gain. They tested SVM three big datasets including the MNIST dataset [LBBH98] with different Bayesian search algorithms, multi-tasking Bayesian search, Random Search and their Fabolas search.

A drawback identified in all these works is, that they only focus on optimizing the cost and the $\gamma$ HPs. Many other relevant HPs for SVM are identified in section 3.4. This was used to formulate search spaces with different numbers of axes in section 3.5.

# 3 Methodology

This chapter aims to give the reader an overview over the developed Hyper-Parameter Tuning setup used in the optimization experiments chapter 4. It lays out the basics, of how a study is set up using this framework. Furthermore, it gives insight, how the experiments were designed. This includes the selection of the datasets, the definition of the search spaces, and setting a timeout for each trial.

## 3.1 Dataset preparation

Recall the definition of the dataset Equation 2.8. A data feature can be defined as a column of the data matrix $X$. So we can define a feature $i$ of the dataset as,

$$\vec{x}_{fi} = \begin{bmatrix} x_{0,i} \\ \dots \\ x_{s,i} \end{bmatrix}. \tag{3.1}$$

In literature, a feature may also be referred to as a *predictor* [KK20]. Usually, raw is a problem for ML algorithms. One problem would be, that outside of text-specialized algorithms, most algorithms work with numeric data only. Another common issue would be, that these algorithms tend to work better with normalized data [AMS+21]. Normalization is discussed in subsection 3.1.4. Besides data scaling, all other data preparation strategies fall under the research domain of *feature engineering*. To enable the development of a HPO framework, that works with common Comma-Separated Values (CSV) tables, some of these issues need to be considered. The basis of this discussion is [KK20].

### 3.1.1 Types of data

Each feature $i$ can hold only one type of data at a time. There are only two broad categories of data, *numeric* data and *categorical* data. While one can identify several different types of numeric data, such as integer numbers, or floating point numbers, for practical purposes they are all considered as floating point numbers. Categorical data are usually strings, the number of different categories per categorical feature should be small compared to the number of samples. Each numeric feature may only contain real scalar data, vectorized data or imaginary numbers would have to be separated into one feature per scalar value. The same consideration applies for categorical data, which must not be concatenated.

### 3.1.2 Missing data

Many algorithms, including SVM, do not work with undefined or missing data. In the context of a numeric feature, this could be a *NaN*[1], $\pm\infty$, or any other non-numeric data. Usually, such data would be discarded. Kuhn and Kjell identify alternative ways, of how to deal with missing numeric data. One might use some *default value*, such as 0. This approach is very use-case specific. For example, if the data is on an exponential scale, changing undefined data to 0 would not allow the feature to be log-transformed later on. A final method would be, to *impute* the missing data. In this process, the feature with the missing data would be used as label for a simple model, such as k-NN. This model would then predict the missing value. Compared to the other two solutions, this is a non-trivial process.

For categorical data, only an empty string may be defined as missing data. In some cases, it is advantageous to encode a missing value, rather than remove the data [KK20, cpt. 8]. Kuhn and Kjell describe, that the fact that a value is missing, can itself be used as valuable information. For example, a dataset dealing with *daily commute* may contain a feature describing the *car manufacturer*. In this case, one might expect, that if this value is missing, then the person might not own a car. Removing such data will likely result in reduced model performance. It is clear, that such considerations are use-case specific. For the PLSSVM-HPO framework, the decision was made, to implement the simpler approach of removing missing data. This decision was reached, because the framework is primarily designed to be used as a tool to do HPO, and the decision, of how to deal with such a problem is highly use-case specific. Furthermore, this does not prevent the user from applying the described preprocessing steps themselves. For example, if a user wants to encode missing categorical data, they can overwrite it with a category called *missing*.

To remove missing data, Kuhn and Kjell identify two possible approaches. A sample with missing data can be removed from the dataset. Alternatively, if too many entries of a feature are missing, then one may consider removing that feature instead. This is due to the fact that while this removes information from all remaining samples, it also reduces the number of samples, that would have to be discarded. In the developed HPO framework, this is implemented by using a removal threshold, that defaults to 10%. If a single feature has more than 10% of it's data missing, then the feature is removed. Thus, the choice of threshold is a trade-off between the loss of information per sample and the reduction of the number of samples. This trade-off also depends on the importance of the feature itself. If the information represented by a feature is low, then the removal threshold can be lowered.

### 3.1.3 Numerical conversion

Numerical conversion describes the method, of converting categorical data to numerical values, that a SVM can use. Two common approaches to conversions are the *hash-based* conversion, or the *one-hot* conversion. In a hash based conversion, the category is computed into a number using a *hash function*. A hash function is a tool used in cryptography, that converts a *key* (in this the category) into a unique integer. As noted, the SVM works best with scaled data. For this reason, the hash function has been implemented by giving each unique category an increasing number starting

[1] This refers to *not any number*, which is often used to encode undefined real numbers.

from 0. This way, if the feature has $m$ different categories, then the hash function would convert them to a range of [0, $m$-1]. They are ordered by the number occurrences. A category that occurs more often will have a lower integer value.

A drawback of such a conversion would be that would order the categories arbitrarily. This may result in a dataset, that is harder to separate. For this reason, one may choose to use an approach, such as one-hot encoding instead. In this scheme, the processed dataset will have one feature per category. Kuhn and Kjell call these *dummy data*. For this categorical data entry, the dummy data column, that matches the entry is 1 and all other dummy data columns of this categorical data are 0. This makes the data more sparse, which may help with separability. Both approaches are implemented in the PLSSVM-HPO framework.

### 3.1.4 Linear transformation

As noted, training on unscaled data can lead to a less optimal model [AMS+21]. This raises the question of which transformation should to use. A common choice is a linear transformation. Within linear transformations, there are a variety of different types. The simplest ones are the min-max scaler. Two of them are the *uniform scaler*, which scales the feature to the interval $[-1, 1]$, while the *unit scaler* scales the data to the interval $[0, 1]$. A shortcoming of the min-max approach may be, that if the feature contains outliers, they will make up the interval boundaries, while the actual data will be compressed somewhere in between. Another common approach is to use a *standard scaler*, which refers to the standard normal distribution. The data is scaled in a way so that it's mean is at 0, while it's standard deviation is 1. This can make this type of scaling more robust against outliers, since the outliers should be much larger than the standard deviation. Besides standard scaling, there are other methods of dealing with outliers, such as clipping and there are many possible combinations of preprocessing steps one might wish to implement. Using standard scaling, the feature $\vec{x}_{fi}$ is scaled by using

$$\vec{x}^*_{fi} = \vec{x}_{fi} \frac{1}{\sigma_{fi}} - \frac{\overline{x}_{fi}}{\sigma_{fi}} \quad \forall i. \tag{3.2}$$

Here, the $\overline{x}_{fi}$ refers to the mean of $\vec{x}_{fi}$ and $\sigma_{fi}$ refers to its standard deviation. While this form may look different, from most definitions of this formula, it highlights the building blocks of the linear transformation: The *factor* $\frac{1}{\sigma_{fi}}$ and the *offset* $-\frac{\overline{x}_{fi}}{\sigma_{fi}}$. Both are calculated on the training dataset using the specified scaling type and then used to transform each feature in the training dataset and each feature in any data, that is to be predicted with the trained model.

## 3.2 Dataset selection

In this project three different datasets with different applications were chosen. The simplest dataset is the *Software-Defects* dataset from Kaggle [WR23], also referred to as *software-faults*. It consists of 21 features and has two unique labels. All features are numeric, some are on a zero-inclusive exponential scale. Given these constraints, users are likely to choose a SVM. The second dataset selected is the MNIST developed by [LBBH98, cpt. III.A]. This dataset is from the field of pattern recognition. It has 784 feature, which are $28 \cdot 28 pixel$ grayscale images of handwritten digits. The

labels are the digits 0 - 9. For this dataset, users are likely to choose a Neural-Network architecture [LBBH98, cpt. II], [SSP03, cpt. 3]. Although the model performance of SVMs may somewhat lower, then other models, LeCun et al. note, that a SVM has the advantage, that it does not require any a priori knowledge of the problem to develop the model [LBBH98, cpt. III.C11]. This may make the SVM easier up for the user to set. The final dataset is the Deepsat Sat-6 dataset [BGM+15]. Like MNIST, Deepsat is an image recognition dataset using $28 \cdot 28 pixel$ images. Unlike MNIST, each pixel consists of a red, green, blue, and near-infrared channel each, resulting in 3136 features per sample. Furthermore, Deepsat does not classify shapes, but rather ground features captured by an observation satellite. Although the use of SVM is uncommon [LBG+20, cpt. 4.2], the PLSSVM library is able to generate a competitive model in a short time, as shown in Table 4.11. All three datasets have in common, that the number of samples is sufficient, to fill a reasonable amount of memory on the GPUs used. This is important to make each study GPU bound so that the energy analysis in section 4.1 is consistent.

### 3.2.1 MNIST

The MNIST dataset [LBBH98] is a well-known choice as a classification benchmark. It consists of images of the handwriting of Bureau of Census employees and handwriting of college students. These images make up the training set of 60,000 images and test set of 10,000 images in equal parts. The training set corresponds with $D_{optimize}$ in this project, while the test dataset corresponds with $D_{holdout}$. LeCun et al. applied preprocessing to the original black and white pictures. First, the images were cropped and centered to $20 \cdot 20 pixel$. To make the images more suitable for classification algorithms antialiasing was applied. This caused the Boolean edges to become blurred. The pixel became grayscale and the dimensions changed to $28 \cdot 28 pixel$ per image.

Each of the best performing SVMs on the MNIST dataset uses additional preprocessing steps. These include *deskewing* and virtual sample contruction. The latter approach constructs new samples are constructed from existing samples. This can be done, by slightly shifting or blurring the edge of the digit. Other ways to create virtual image samples are to mirror the image, which would not work with digits, or rotating images, which would make classification harder for SVMs. While adding virtual samples can slightly improve the accuracy, hardware limitations prevented the use of a much larger dataset. Therefore, this approach was not used.

#### Deskewing MNIST

When using photographed or scanned images, then they may be skewed. This means, that an image may be stretched, rotated, or translated, or any of those combination of these. Many algorithms, including SVMs, use one-dimensional data. To generate one-dimensional data from two-dimensional data, w.l.o.g. the rows of each sample are concatenated. This makes these algorithms more sensitive to skewed images, because a skewed image shifts the data from one feature to another feature, compared to an unskewed image. In the deskewing process, each digit is tilted upward and positioned using its center of mass. Deskewing was done using a lightweight procedure [WGL16] which is based on the theory of *principal component analysis* [Shl14]. In this approach, the written numbers can be thought of as a two-dimensional random distribution. Each pixel $\neq 0$ is a sample, from the distribution. Calculating the covariate of this distribution, can be used as an estimate of the rotation angle of the digit. An affine transformation is used to rotate and

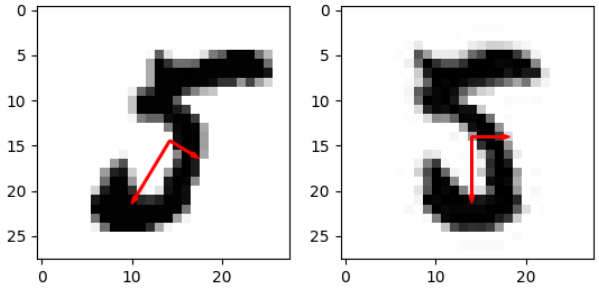


**Figure 3.1:** An example of deskewing images of MNIST. The arrows are defined over the covariance matrix of the digit's distribution. Eigenvectors of the matrix define their directions and the square-root of the eigenvalues define their lengths.

center the image. Figure 3.1 visualizes this process. Affine transformation for digit recognition was used before in [WKT01], although their Optical Character Recognition implementation used Cross-Correlation to determine the digit. Deskewing improved the SVM accuracy by about 0.5% without increasing the size of the dataset. Therefore, the small additional overhead for each prediction was considered reasonable.

#### QMNIST

It would be remiss not to mention recent critique of the MNIST dataset. Using the original MNIST dataset may result in reported model performance that is somewhat inflated compared to reality [YB19]. In their findings, they suggest to increase the size of the MNIST test dataset, to obtain more realistic performance results. While this criticism does not seems unfounded, the goal of this work is not to find the most realistic performance numbers. The main goal is to compare the models found by each optimizer with each other, and a secondary goal is, to compare the found model performances with existing literature. According to Yadav and Bottou, classifier ordering model selection should still be reliable. Thus, the main goals are met and the use of MNIST allows for more comparisons. This made MNIST the preferred choice.

### 3.2.2 Deepsat Sat-6

The main motivation behind Deepsat [BGM+15] was to develop a dataset, that could serve as a high quality source of satellite imagery, tailored for testing ML algorithms. These images were obtained from [US 09]. Basu et al. developed two separate datasets, Sat-4 with four labels and Sat-6 with six classes. The latter was used in this work, it may be referred to as Deepsat. The class labels of Deepsat Sat-6 are: barren land, trees, grassland, roads, buildings, and bodies of water. Each image is cropped to $28 \cdot 28 pixel$. With a ground resolution of 1 pixel/m, each image spans about $28m \cdot 28m$. The dataset itself is consists of 324,000 training images and 81,000 test images. For this work, only a quarter of the training set was used for $D_{optimizer}$, while $D_{holdout}$ consists of all test images. The number of training samples was reduced so that the PLSSVM model fits into the used GPUs Video Random-Access-Memory (VRAM)used . This was done for performance reasons. Using the full dataset would require the PLSSVM to use the implicit CG-solver, instead of the

CG-explicit. Due to the complexity of the dataset, using the implicit solver would take significantly longer, fitting each model takes an average of 1000 iterations of CG. The enormous increase in the time required by the implicit solver would make any detailed analysis impossible.

## 3.3 PLSSVM-HPO framework

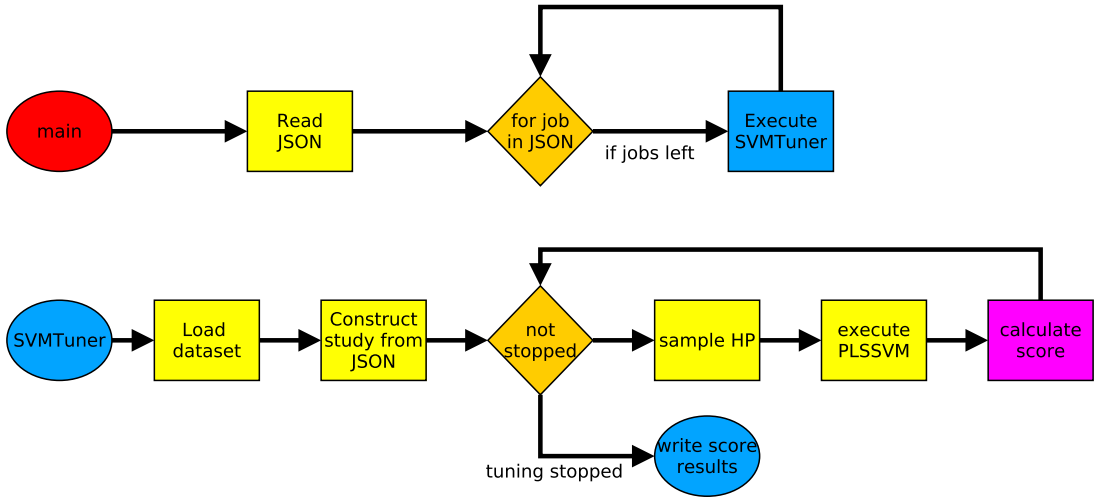



**Figure 3.2:** PLSSVM-HPO flow diagram

For this project, a user tool has been developed to support searching HPs using Grid Search, Random Search and TPE optimization (subsection 2.4.4). On one end, the user defines the tuning parameters, the dataset and the HP search space. Then the framework searches the HP space and finds the best model under the given constraints. Figure 3.2 visualizes a high-level sequence diagram of the framework. As indicated in chapter 1, PLSSVM [VCBP22] is used as the SVM implementation and Optuna [ASY+19] as the HP sampler. The framework itself is written in Python3.10 [Ros95], it loads the study parameters, the dataset and then runs the study using Optuna and PLSSVM. Unlike most existing HPO frameworks, this tool provides the user with a text file based frontend. This eliminates the need for programming to set up a tuning study.

### 3.3.1 DatasetContainer class

The Dataset Container is a data structure responsible for loading the dataset used in a study. A diagram of the DatasetContainer class can be found in Figure 3.5. It can load any CSV table, convert all data to numeric form, and clean up missing, or incorrect entries. To optimize loading time and dynamic memory allocation, a raw format has also been developed. The raw format can only load a purely numeric CSV table. As such, the header has to be removed as well. For large datasets, this can drastically reduce the load time. To optimize load times, a user can convert a normal dataset and save it in a raw format using the Dataset Container file on the command line. Other dataset types, such as the sparse libsvm format, are planned but not yet implemented.

When the dataset is loaded, the user is able to split a Dataset Container into two smaller Dataset Container. The dataset is shuffled and then the selected size is taken from the top and used to create a new Dataset Container. This feature is used when the user does not select a separate validation holdout dataset in the study settings. In either case, each study has two Dataset Container objects. One is is the optimizer dataset, which is used to train and select a model. The other is the holdout dataset, which is used to measure the final model performance unbiased by the optimization process.

To use the Dataset Container as an optimizer dataset, the following functionality has been developed: Given a Hyperparameter object, the training data is prepared. This includes slicing the dataset using the holdout method, or CV described in section 2.2. The next step is to perform any selected preprocessing step on the current training slice of the dataset. The resulting training slice is used to train the model. Then, the same steps are repeated for the validation slice of the optimizer dataset. The performance of the validation slice is measured, it is one of the most important metrics, that may be used for model selection. These preprocessing steps are saved to a file so that the end user can recreate them for prediction samples.

In order to use the Dataset Container as a holdout dataset, the functionality has been implemented, to preprocess the data and measure performance. In this mode, a preprocessing file is loaded. Each step specified in the file is performed on the entire dataset. This dataset is then used to measure the performance of the found model. Due to the separation this should result in less bias in the reported performance, as described in section 2.2.

### 3.3.2 Hyperparameter class

This is a class that contains all the HP that a user can select. Setting any of the object's attributes will be checked for type and range. For example, the degree parameter must be an integer, or integer string and the number must be greater then 0. These checks are implemented and enforced through the use of asserts. Setting an attribute incorrectly will raise an *AssertionError*. This has the consequence, that the user must ensure that the type and interval of a search axis for a given HP are compatible with that HP. The reason for such an approach is to reduce the risk of undefined behavior in any component that uses the Hyperparameter object.

Any HP, that is not set in the constructor will be set to a default value. In each trial, the sampler in the SVMTuner study object (subsection 3.3.6) sets each attribute, that is specified in the study entry. At this point, the Hyperparameter object can be used to set the PLSSVM and perform preprocessing on the dataset.

### 3.3.3 Hyper Parameter Sampling

To set each HP attribute of the Hyperparameter object, the sampler draws all HP of the defined search space. In each case, first a Hyperparameter object with default values is constructed first.

### Continuous sampler

The group of continuous samplers includes the Random Search and the TPE optimization, both support the use of continuous search spaces. Recall from subsection 2.4.3 that each search space $\Lambda$ is made up of search axes $\Lambda_a$. In the first step, all conditional search axes $\Lambda_{ca}$ are removed from the overall search space $\Lambda$. Then the sampler iterates over all the remaining axes of $\Lambda$ and draws a HP for each. For each HP the corresponding attribute of the Hyperparameter object are set. After that, the conditional axes $\Lambda_{ca}$ can be evaluated. For each conditional axis, the relevant HP attribute of the Hyperparameter object is checked. If the relevant attribute is set, then a HP of the conditional axis is drawn and set. Otherwise, the conditional axis is ignored. For example, drawing the degree HP would be redundant, if the kernel was determined to be linear. While evaluating unused conditional axes would not make no difference for the Random Search, it would unnecessarily increase the complexity of the TPE model.

### Grid Search

Optuna's Grid Search is implemented, to draw a random untested combination from the search space. They ensure that Optuna will try all available combinations of the search space exactly once, unless the study is aborted earlier. However, the same conditional sampling approach described above is incompatible with Optuna's Grid Search. To quote the website, a conditional search using the Grid Search *works, but inefficiently*[2]. If an unmodified Grid Search would be performed, Optuna would draw all possible combinations of the search space. To distinguish them, this approach is referred to as the *unconditional grid*. This grid includes irrelevant conditional combinations, such as combining a linear kernel with different polynomial degrees. Using a grid in this way would increase the number of trials needed to perform an an exhaustive search. Therefore, the performance of the Grid Search would artificially suffer, compared to the continuous sampler methods.

In the worst case scenario, the number of trials would be almost three times the conditional combinations. Suppose a search space of

$$\Lambda = \Lambda_k \times \Lambda_d,\ \Lambda_k = [linear, poly, rbf]\,, \tag{3.3}$$

and the degree search axis $\Lambda_d$. $\Lambda_d$ is dependent on the polynomial kernel only. The number of unconditional combinations is $||\Lambda||_u = 3 * ||\Lambda_d||$, whereas the number of relevant conditional combinations would be $||\Lambda||_c = 2 + ||\Lambda_d||$. For

$$\lim_{||\Lambda_d|| \to \infty} \frac{||\Lambda||_u}{||\Lambda||_c} = ||\Lambda_k|| = 3. \tag{3.4}$$

This example is also true, if search space $\Lambda_d$ was substituted by another space, that was conditional on only one of the kernels. If $\Lambda_d$ was replaced by a space that was partially or completely conditional on two or more of the kernels, then this number would be lower than this worst-case scenario. This is, what happens in practice, in which case this number tends to be around two. The simple proof holds only for the SVM search spaces, since all conditional parameters only depend on the choice of the kernel.

[2] https://optuna.readthedocs.io/en/stable/reference/samplers/index.html [checked 11.04.]

To avoid this issue, a *conditional grid* search has been developed. In this mode, the grid is flattened before it is used. This means, that each viable combination is precomputed. For the PLSSVM model HPs, all conditional HPs are dependent on the kernel. One conditional parameter is $\gamma$, which is defined for both the polynomial and the rbf kernels. Otherwise, the polynomial kernel uses the *coef0* and *degree* parameters. This divides the search space into three subspaces, one for each choice of kernel. Each subspace is filtered for relevant conditional dimensions. As example, the degree and coef0 axes are removed from the rbf's kernel subspace. Then each subspace is expanded. Starting with the kernel, it is combined with each entry of the first dimension $\Lambda_1$, resulting in the combination vector $\Lambda_{flat}$. The length at this point is $||\Lambda_{flat}|| = 1 \cdot ||\Lambda_1||$. Then each entry of $\Lambda_{flat}$ is expanded with the next axis, which would result in $||\Lambda_{flat}|| = 1 \cdot ||\Lambda_1|| \cdot ||\Lambda_2||$. This process of expanding the subspace is repeated, until no axis is left. At the end, each the three expanded subspaces are concatenated into one long space. The name flat comes from the fact, that it is a long one-dimensional vector. Each entry of the flat grid is a combination of different scalar HP values. This is implemented as a dictionary, using the name is used as the key for each HP. Sampling from the grid would be done by drawing an index of the vector. The entry at that index is then used to evaluate the objective function.

As example, let

$$\Lambda = \Lambda_{kernel} \times \Lambda_{degree} \times \Lambda_{cost} \tag{3.5}$$

$$\Lambda_{kernel} = [k = \text{"}linear\text{"}, k = \text{"}poly\text{"}], \quad \Lambda_{degree} = [d = 2, d = 3], \quad \Lambda_{cost} = [c = 1, c = 10]. \tag{3.6}$$

1. Now we generate the filtered subspaces $\Lambda_{linear} = [\{k = \text{"}linear\text{"}\}] \times \Lambda_{cost}$ and $\Lambda_{poly} = [\{k = \text{"}poly\text{"}\}] \times \Lambda_{degree} \times \Lambda_{cost}$.
2. Then we expand both subspaces the first time to $\Lambda_{linear} = [\{k = \text{"}linear\text{"}, c = 1\}, \{k = \text{"}linear\text{"}, c = 10\}]$ and $\Lambda_{poly} = [\{k = \text{"}poly\text{"}, c = 1\}, \{k = \text{"}poly\text{"}, c = 10\}] \times \Lambda_{degree}$
3. The polynomial subspace needs one more iteration to be fully expanded $\Lambda_{poly} = [\{k = \text{"}poly\text{"}, c = 1, d = 2\}, \{k = \text{"}poly\text{"}, c = 10, d = 2\}, \{k = \text{"}poly\text{"}, c = 1, d = 3\}, \{k = \text{"}poly\text{"}, c = 10, d = 3\}]$
4. With both subspaces fully expanded, they are concatenated, which yields the flat-grid $\Lambda_{flat} = [\{k = \text{"}linear\text{"}, c = 1\}, \{k = \text{"}linear\text{"}, c = 10\}, \{k = \text{"}poly\text{"}, c = 1, d = 2\}, \{k = \text{"}poly\text{"}, c = 10, d = 2\}, \{k = \text{"}poly\text{"}, c = 1, d = 3\}, \{k = \text{"}poly\text{"}, c = 10, d = 3\}]$
5. Notice, the flat-grid $\Lambda_{flat}$ has only six combination, whereas the unconditional grid $\Lambda$ has 8 combinations.

A possible disadvantage of the precomputed grid is, that for large search spaces, computing all possible combinations may exceed practical runtime and memory constraints due to the curse of dimensionality. This is avoided by setting a pragmatic upper limit of $10^6$ combinations calculated for an unconditional grid[3]. The number of $10^6$ was chosen arbitrarily because a grid of this size cannot be searched efficiently. For this reason, both grid methods are available to the user. In this work, only the conditional grid was tested.

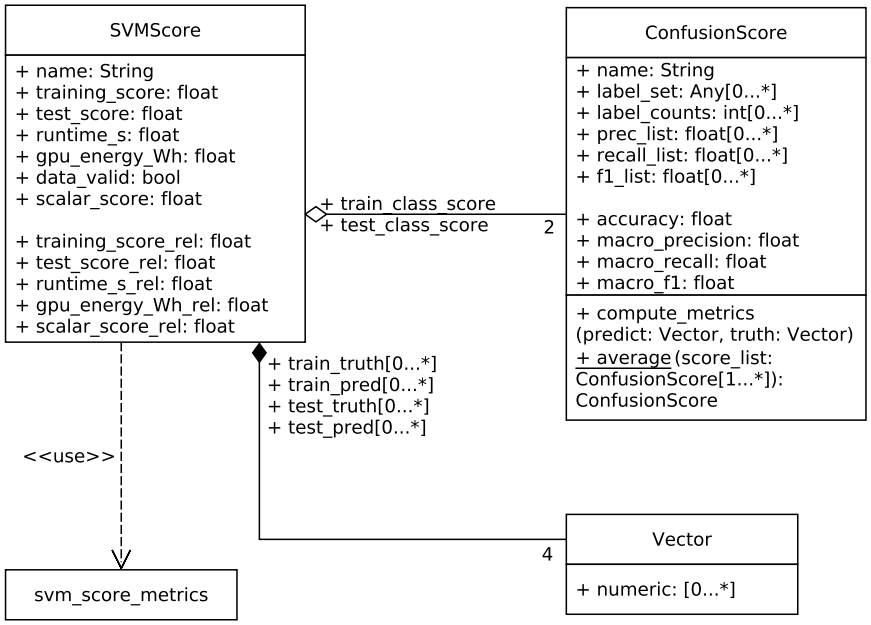



**Figure 3.3:** Class diagram of SVMScore, the Vector class refers to a one-dimensional NumPy-array [HMW+20]. Using holdout validation, each list only contains one entry. If used with CV, each list has k-entries, one for each fold.

### 3.3.4 SVMScore class

To select one of many models requires insight into the model's performance. For this purpose, the *SVMScore* class, seen in Figure 3.3, has been developed. It is mainly used as a container, for all relevant different metrics of the model. An overview of the classification metrics is given in section 2.1. It contains a score for each the training and validation score of $D_{optimizer}$ (section 2.2) called *training_score* and *test_score* respectively. The *training_score* is defined as the *train_class_score.accuracy*, since the framework is mainly designed to tune classification models. However, it is conceivable that these two scores could store any regression metric, if Support Vector Regression was to be implemented. A number of attributes are not related to the performance of the model, but to the resources used to generate the models. Two of the most important ones are the *runtime_s* and the *gpu_energy_Wh*[4], both are logged, while the model is being fitted. Of other logged data (e.g. CPU or Random-Access Memory (RAM) utilization) the *min*, *max*, *mean* and *standard deviation* is calculated and stored in the SVMScore object. If an NVidia GPU is selected, this list is extended by the CUDA core utilization and the VRAM. The *data_valid* attribute encodes whether the score is usable. This may be relevant in cases where fitting the model crashes or a hard timeout occurs, in which case no model and no predictions would be available. Two attributes of the class *ConfusionScore* are used to keep track of all the model performances for each label and the aggregated classification scores. It's main usage is, to compute all the metrics and to average multiple ConfusionScore in case of CV using the arithmetic mean. For each of the float attributes

[3]This can be calculated trivially, by multiplying the length of each HP axis.
[4]As of now, the *gpu_energy_Wh* is only defined if a NVidia GPU was selected.

of the main *SVMScore* object, there exist a *-_rel* attribute. This is calculated by using the first valid result as the baseline SVMScore-object and then calculating the relative value of the corresponding attribute of both objects.

The most important attribute is the *scalar_score*. It is used by the tuner, to select the model and it is used as feedback for the TPE optimizer. To set this attribute, the SVMScore object must be converted to a scalar value. This is done using one of the *metric functions* implemented in the *svm_score_metrics* file. Each of the metric functions takes the SVMScore object as argument and returns a single attribute of the SVMScore, or an aggregated score. A relevant example is the *test_acc*-metric $test\_acc(score) = score.test\_score$. It is conceivable, that one could implement any other metric. For example, one that uses the *train_class_score.macro_f1* in case, an unbalanced dataset is to be evaluated (see section 2.1, or [GBV20, cpt. 4.2]). Recall the definition of the metric function $f1_j$ in Equation 2.3: If the number of true positives of any classifier $j$ is zero, then the resulting $f1_j$ is undefined. This can lead to undefined *scalar_scores*. Returning an undefined float does not work with the Optuna library. So any undefined value must be converted. Since the metric functions are defined from a range of 0 to about 1, any undefined value will be encoded as -1.

Another possible type of metric function is a multi-objective metric. One implemented example is the *time_metric*

$$time\_metric(score) = score.test\_score * 0.8 + rel\_trafo(score.runtime\_s\_rel) * 0.2 \tag{3.7}$$

$$rel\_trafo(x) = \frac{2.0}{1.0 + x}, \tag{3.8}$$

where the score is weighted towards the performance. However, models with less runtime are are rewarded. The relative transformation is implemented to ensure, that the transformed values remain between 0 and 2. Furthermore, $rel\_trafo(x) \to 1$, for $x \to 1$. It should become evident, that there are many possible metrics that a user might want to try, depending on the use case. For this reason, any user is invited to write their own metric function in the *svm_score_metric* file. Any of the described attributes can be used.

### 3.3.5 PLSSVM trial execution

With the dataset loaded and the HPs sampled, the study can proceed to evaluate the objective function. An overview of the following is given in Figure 3.4. Before fitting any data, a telemetry thread is spawned. The purpose of this thread is to log hardware information, such as the CPU utilization, or the GPU power consumption. It logs hardware information every second it is not stopped. The GPU energy usage is calculated by integrating the power over the scheduling interval.

While the telemetry is being logged, the trial can fit the dataset $D_{optimizer}$ using PLSSVM. If CV is not used, k is set to 1 and data is fit only one time. Additionally, for k=1, the size of the training and validation slices of $D_{optimizer}$ can be configured using the Hyperparameter object. Using CV will fit k models with the same HP, but k different training and validation slices. Each time a model is is fitted, a new PLSSVM thread is spawned. The rationale of this decision is, that it allows to implement a *hard timeout*. That is, if a model does not converge, or converges too slowly, then the

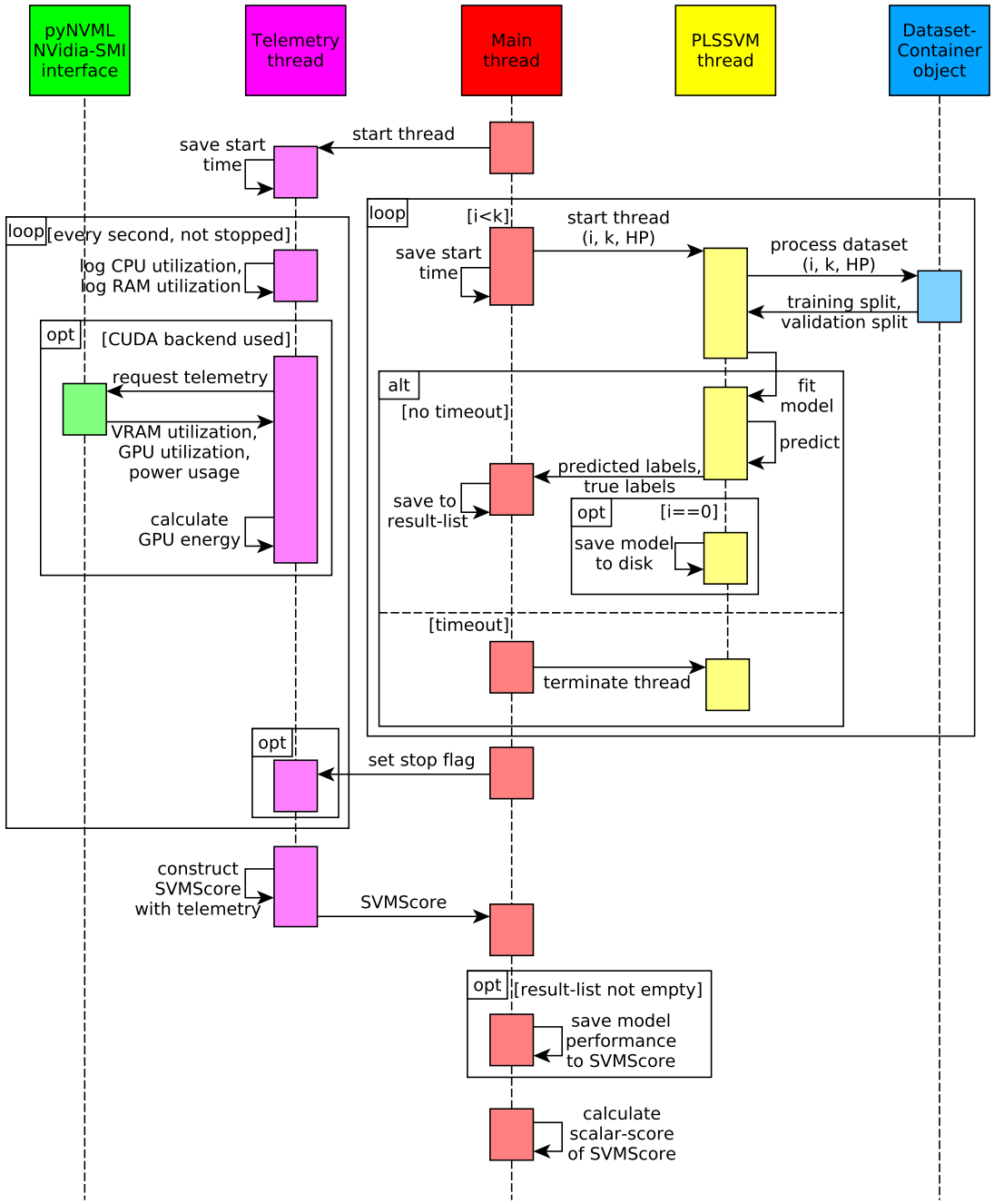



**Figure 3.4:** Sequence diagram of fitting and evaluating a PLSSVM model. If CV is used, then the main loop is executed k-times. Otherwise it is executed only once.

trial thread may terminate it[5]. Compared to the hard timeout, a user can set a *soft timeout*, by setting the *max_iter* HP. This parameter sets the limit of iterations of the CG algorithm used by PLSSVM,

[5] A hard timeout is not implemented in PLSSVM as of now.

as described in subsection 2.5.3. When used on the same hardware, the same dataset, the same multiclass classification type and the same type of CG solver, then the time of each CG iteration is very consistent. Therefore, executing a fixed number of CG iterations takes about the same time.

After the model is fitted, it predicts the samples of the training and validation slices, so that the model performance can be evaluated. When the main loop exits, the main thread sets the stop flag of the telemetry thread. This allows the telemetry to gracefully exit, construct an SVMScore object, and set it's telemetry data, as described in subsection 3.3.4. Then the SVMScore is returned to the main thread. If at least one model was fit, the model performance is evaluated and saved in the SVMScore object. In case, CV is used, all k-models are used to evaluate their performance. Their performance is then averaged. In all cases, the first model fit during the trial is saved to disk for later use. With the model performance calculated, the SVMScore can be converted to a scalar score using the selected metric in *svm_score_metric* (see subsection 3.3.4). The scalar score is then returned by the trial.

### 3.3.6 SVMTuner class

The *SVMTuner* class is the main framework, that combines the classes discussed so far, as can be seen in Figure 3.5. For each defined study, an SVMTuner object is constructed. During construction it loads all the study settings (subsection 3.3.7) and the two datasets. When finished, it can be run using the *run_job* method. This will repeat a loop of sampling HPs, executing the PLSSVM trials on the *optimizer_dataset*, and calculating the score of the trials. If at least one stop criterion is met, this loop is stopped and the results and the model are saved to a file. This cycle can be seen in Figure 3.2.

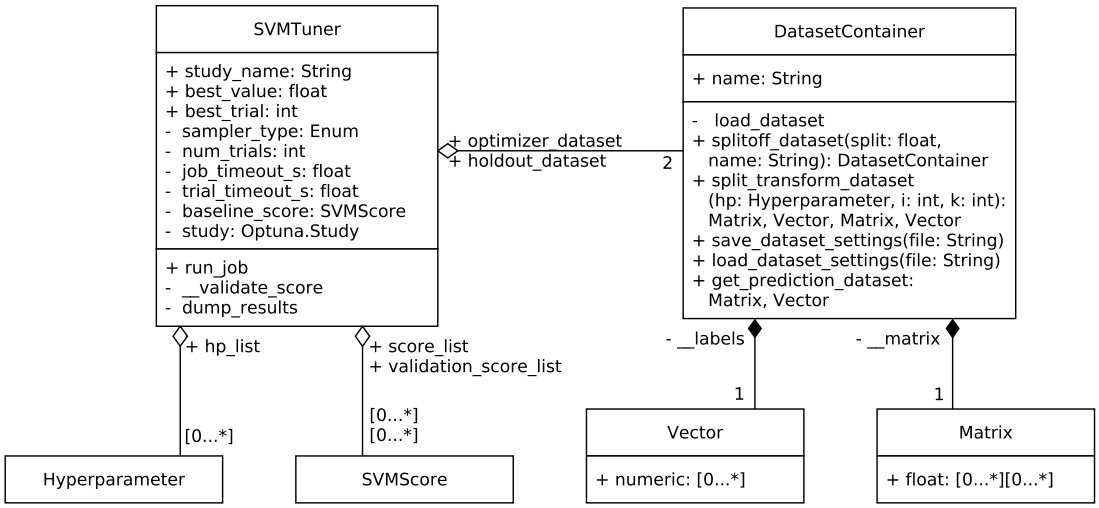



**Figure 3.5:** High-level UML-Diagram of the SVMTuner class. The DatasetContainer class is described in subsection 3.3.1, whereas the Hyperparameter container is described in subsection 3.3.2. Vector and Matrix refer to a one- and two-dimensional NumPy-array respectively [HMW+20].

The *best_trial* and *best_value* attributes are the trial number with the best scalar_score and score itself (subsection 3.3.4). With the *sampler_type* enum attribute, the tuner object selects one of the implemented Optuna sampler backends. To keep the diagram compact, only the two most important stop criteria are listed. The *num_trials* sets the number of trials the study will execute. If it is set to exactly 0, then the study will be skipped and the datasets are not loaded into memory, to save time. Another important stop criterion is the *job_timeout_s*, which determines how long the study is run. At least one of these two stop criterion must be set. An exception is the Grid Search, where num_trials is set to the number of unique combinations, if it is not specified. The hard timeout (subsection 3.3.5) of each trial of a study can be set with the *trial_timeout_s* attribute. The *baseline* score is a copy of the first valid SVMScore object, which is used to calculate relative performance scores (subsection 3.3.4). *hp_list* is a list of the HPs used in each trial. The *score_list* contains the SVMScore object of each trial, evaluated on *optimizer_dataset*. Parallel is the *validation_score_list* attribute, which is not used by default. When used, it stores a SVMScore of the performance of each fitted model on the *holdout_dataset*. Tuning is done exclusively using the SVMScore object associated with the optimizer_dataset. When a study is complete, both lists are dumped into a CSV table. These results are used for this report to provide insight into the tuning performance.

### 3.3.7 Tuning job Input data

Each study is defined using a metadata file in the JSON format. This file format has the advantage, that it can be trivially loaded into a Python dictionary. An overview of the file structure is given in Figure 3.6. The *CommonSVMInfo* entry applies to all studies defined as *SVMJobs*. Both the CommonSVMInfo entry and the hyperparameter entry must use this exact name, which is defined as their dictionary key. Each SVMJob entry only requires the String *SVMJob* to be contained in their name. However, each SVMJob name has to be unique. Using SVMJob in the name ensures, that studies defined as SVMJobs are executed.

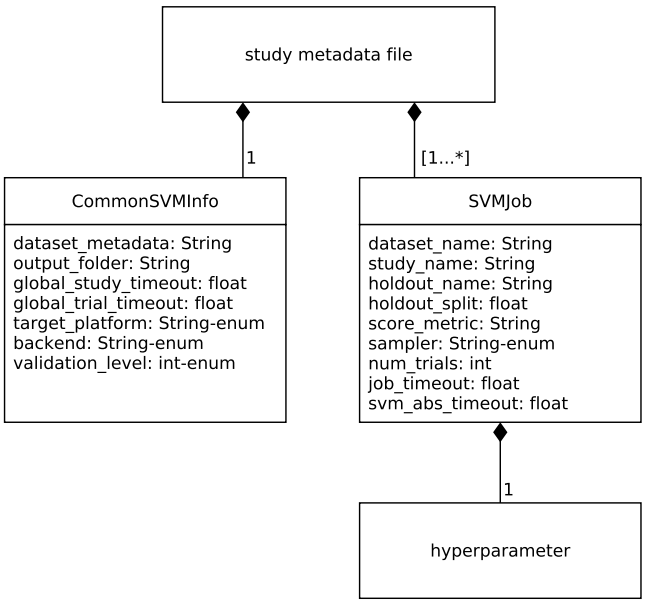



**Figure 3.6:** Relational diagram of input JSON metadata file.

#### CommonSVMInfo

The *dataset_metadata* field is required, it contains the path to the dataset- etadata file. Each key of the dataset-metadata is the name of a dataset, it's values are instructions, on how to load the dataset. With the *output_folder* field defines the path where all studies will save their output files (see **??**). Selecting the PLSSVM target-hardware and backend can be done using the *target_platform* and *backend* fields respectively. If not selected, both default to *automatic*, in which case PLSSVM will automatically select the best available choice. *validation_level* refers to, how the model is validated on the holdout dataset. It can be set, to do no validation, validate only the best model or validate all models. If not set, it defaults to the second choice.

#### SVMJob

On the SVMJob entry, the *dataset_name* refers to the dataset entry in the dataset metadata file. This dataset will then be the optimizer dataset of the later generated SVMTuner object. To load the holdout dataset, either the *holdout_name* field, or the *holdout_split* field may be used. The first one loads a separate dataset, while the second one splits the specified percentage from the optimizer dataset (see subsection 3.3.1). Exactly one of these fields must be set. The *score_metric* field specifies which of the metrics in the svm_score_metric file to execute (see subsection 3.3.4). If not set, it defaults to the *test_acc* metric, which reports the accuracy of the model on the test slice of the optimizer dataset. If set, it must correspond with one of the metric functions in the svm_score_metric file, to prevent arbitrary code execution. Setting the *num_trials* field sets a limit on the number of trials, that are run, which is described in subsection 3.3.6. To set the study timeout of the SVMTuner, one can use the *global_study_timeout* field in CommonSVMInfo. It will set the job_timeout_s of all studies. This can be overridden per study with the *job_timeout* field. If the global_study_timeout remains unset, it defaults to a value of 1h. The hard timeout (subsection 3.3.5) of each trial of the studies is set the same way using the fields *global_trial_timeout* and *svm_abs_timeout* respectively.

#### hyperparameter

In each hyperparameter field, the user defines each HPs search space by setting the *key* with the name of the HP. The value of the parameter can be set as a *single value* or a *list of values*. If the parameter is a numeric value, then a range can be set using an invocation written as string. An example invocation is "float(0, 1)". Here, the first argument is the lower limit and the second argument is the upper limit of the search axis. The upper limit is included in the search axis, it must be equal to or greater than the lower limit. This example would result in a search axis of $[0, 1]$. For Grid Search, the invocation requires a discretization, which is set with an additional *steps* argument. If set, it defines how many equidistant values are generated in the interval. An invocation of "float(0, 1, steps=3)" would result in the search grid $\{0, 0.5, 1\}$. The steps argument may not be used with any continuous sampler[6]. Using an all equidistant space may not be suitable for values approaching 0 or very big values. In this case, the scale can be changed to logarithmic,

[6] Using discrete spaces for continuous sampler is not their intended use-case (subsection 2.4.4) and as such it is not fully implemented as of now.

by supplying the argument *log*. An example would be "float(1, 1000, steps=4, log=True)", which would be interpreted as the grid {1, 10, 100, 1000}. On the log scale, both limits must be strictly greater than 0. The log argument is compatible with all samplers. Alternatively, one can use a integer invocation using "int(0, 3)". The same rules apply, as did for the float invocation. One difference is, that due to the nature of the data, no discretization is required for a Grid Search. This would be translated to a search axis of {0, 1, 2, 3}. Note that due to the integer data, the data cannot be used on a logarithmic scale. This invocation is compatible with all sampler types.

## 3.4 Hyper Parameter Sensitivity tests

To find the range and importance of each used HP used, each parameter was varied with the rest using a default setting. This approach gives only limited insight, as the behavior of each HP is depends on the other HPs. For this reason, each HP has been used with each kernel, as it changes the behavior of the SVM the most. The most important HPs are: Kernel (Equation 2.5.4), which defaults to rbf. Scaling, which defaults to uniform transformation (subsection 3.1.4). Cost defaults to a value of 1, while the tolerance $\epsilon$ (also called tol) defaults to $10^{-8}$. The degree HP is used for the polynomial kernel, it defaults to 3. $\gamma$ used in polynomial and rbf kernel defaults to $1/n_{features}$. The *coef0* parameter has been set to a default of 0 for the significance tests.

As noted in subsection 3.1.4, the scaling of the data is very important for the performance of a SVM. Standard, uniform, and unit scaling perform well on MNIST (Figure 3.7a) and they are popular choices for scaling. Quartile scaling does not perform well on MNIST and on Software Defects (Figure 3b in the appendix). Unscaled data produces bad result on all datasets. Therefore these two scaling types are not considered to be used on the scaling search axis. Additionally, the default choice of uniform scaling does produce suboptimal results on Deepsat, which can be seen in Figure 4a of the appendix. The cost HP needs careful consideration. Setting it too low will result in bad performance, while setting it too high may cause a model to never converges. The default of 1 seems to work well on all datasets and a range of $10^{\pm 2}$ seems to cover all the performance plateaus reasonably well. Setting the degree HP is also a important choice. While any choice seems to lead to a good model on MNIST (Figure 3.7c), setting it too high on Deepsat seems to lead to a bad performance, as can be seen in Figure 4c in the appendix. So a range of 3 to 9 was determined, so it covers the optimum for MNIST. Using the default $\gamma 0$ HP seems to produce good results on all dataset. It is defined as $\gamma 0 = 1/n_{features}$. The correct choice seems to be very important for the polynomial kernel, as can be seen in Figure 3.7d. Here, the $\gamma 0 = 1.28 \cdot 10^{-3}$. While the choice of coef0 = 0 works well in most cases, it seems, that a choice of coef0 = 1 can work even better. This can be seen in the appendix in Figure 4e, where a coef0 = 0 appears to be less robust. $\epsilon$, also known as *tol*, seems to produce very stable results for tol < $10^{-8}$.

How limited the significance tests are can be demonstrated by Figure 4c of the appendix. This plot is part of the significance test and uses the default values described above. It seems, that increasing polynomial degrees results in decreasing model performance on the Deepsat dataset. Yet it was observed, that one TPE study found an optimum on Deepsat for the HPs {kernel=poly, degree=9, coef0=1, cost=6.70}. This underlines the importance of finding a good combination of HPs.

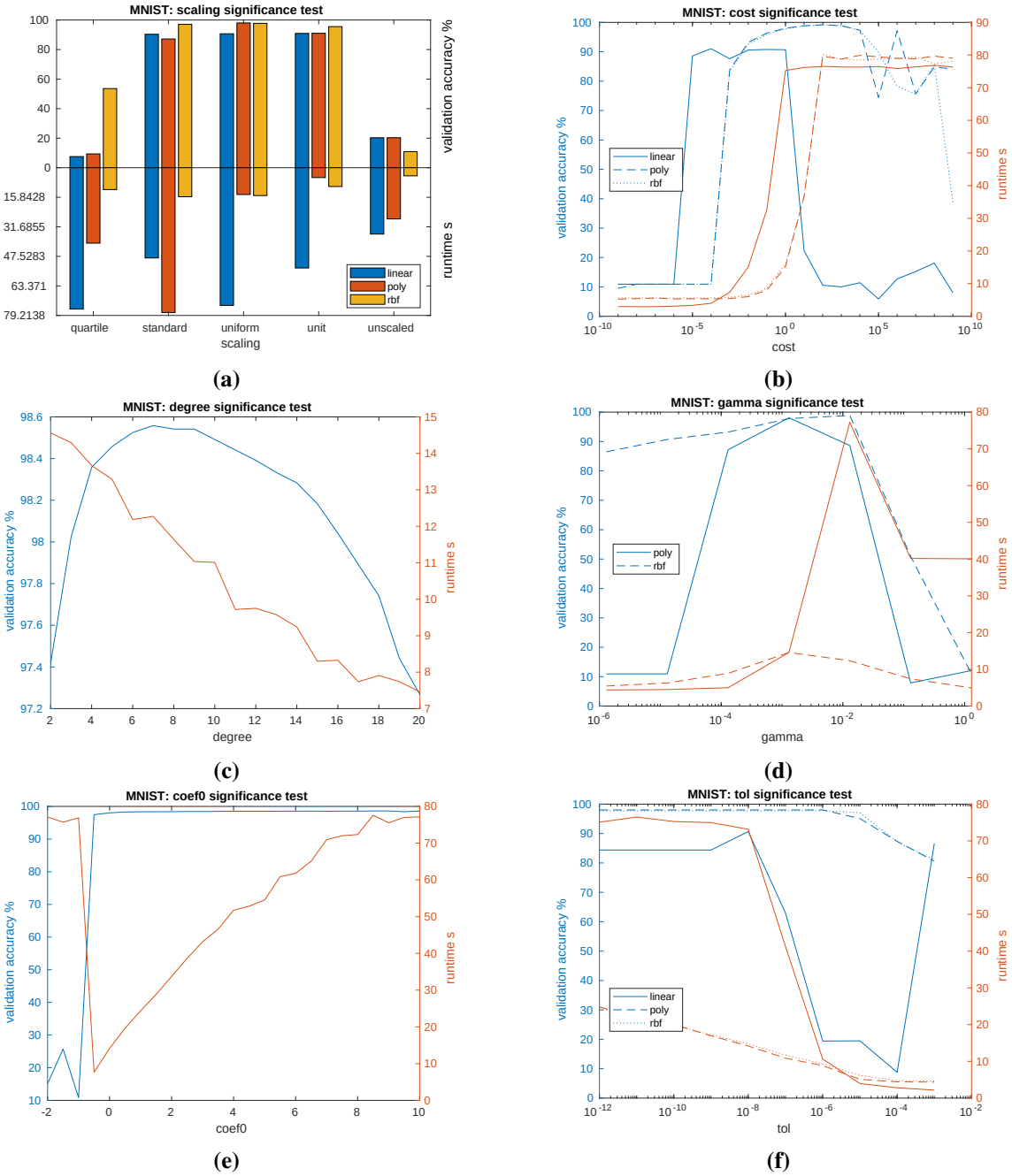



**Figure 3.7:** These plot show the main HPs behavior over the MNIST dataset. Sensitivity tests of the other datasets can be found in Figures 3 and 4 of the appendix.

## 3.5 Search space definitions

This chapter describes the search spaces used in all tests. The premise is, that a user may not know the exact range and the importance of each HP. Therefore, it would not be useful to develop individual search spaces for each dataset. Instead, the results of all the tests in section 3.4 were used to formulate search spaces to be used on all datasets. Each search space adds one additional search axis, with the exception of search space four, which adds two axes. With each aditional axis, the search space increases significantly. The order in which each HP search axis is added is determined by the estimated importance of each HP. While this process is largely guesswork, the rationale is described in each search space definition. The ranges are chosen, so all three datasets work reasonably well.

### 3.5.1 Default values

Before defining the search spaces, one must define the values, that are not in the search space. If not noted otherwise, then the default values were used from PLSSVM [VCBP22] and they were verified to work well with the selected datasets. The values are shown in Table 3.1.

| HP | value | comment |
|---|---|---|
| $kernel$ | "rbf" | |
| $scaling$ | "uniform" | determined by test |
| $cost$ | 1 | |
| $degree$ | 3 | |
| $gamma$ | $\gamma_0$ | $\gamma_0 = 1/n_{features}$ |
| $coef0$ | 1 | 1 yields same or better results then 0 |
| $tol$ [a] | $10^{-9}$ | determined by test |
| $multiclass$ | "oaa" | multiclass classification strategy, defined in subsection 2.5.5 |
| $split\_train$ | 0.8 | Slice of $D_{train}$ |
| $max\_iter$ | 2000, 3300 | 3300 on MNIST, else 2000. Soft timeout determined in section 3.7. |

**Table 3.1:** Default values used. [a] may also be referred to as $\epsilon$.

### 3.5.2 Search space 0

Search space 0 is designed to be a simple baseline. Due to it's definition, it has only nine unique combinations. If a HP is not listed, then it is left as default. According to an analysis about feature scaling [AMS+21], SVM is sensitive towards feature scaling, which was also observed in the tests. It was chosen as the second axis because that in the tests, a HP like $cost$ did always yield good performance for it's default value. This is not the case with the scaling, which had more varied results, depending on the kernel and dataset.

| HP | value |
|---|---|
| *kernel* | ["linear", "poly", "rbf"] |
| *scaling* | ["standard", "uniform", "unit"] |

**Table 3.2:** Search space 0 definition.

### 3.5.3 Search space 1

As indicated to in the previous sections, the *cost* HP is also important. Therefore, it was chosen as the third axis. The float search space invocation is described in subsubsection 3.3.7. This search space has 45 unique combinations and as such, it is described as a *small search space*. In our tests, average computation times of one to two minutes per trial were observed (section 3.7), depending on the dataset and sampler. Thus, calculating all trials of the study will not require many resources.

| HP | value |
|---|---|
| *kernel* | ["linear", "poly", "rbf""] |
| *scaling* | ["standard", "uniform", "unit"] |
| *cost* | "float(1e-2, 1e2, steps=5, log=True)" |

**Table 3.3:** Search space 1 definition. 1e-2 is the programming notation of $1 \cdot 10^{-2}$. The steps argument is left out for continuous sampler in this and all following search space definitions.

### 3.5.4 Search space 2

Figure 4c in the appendix section demonstrates, that the choice of the degree HP is important as well. Since the degree HP depends on the kernel, there are 135 unique combinations on the conditional grid (subsection 2.4.4). In this case, an unconditional sampler would have 315 combinations. With 135 combinations, each study will take some time to compute, but it will still be reasonable. Therefore, this was called the *medium search space*.

| HP | value |
|---|---|
| *kernel* | ["linear", "poly", "rbf"] |
| *scaling* | ["standard", "uniform", "unit"] |
| *cost* | "float(1e-2, 1e2, steps=5, log=True)" |
| *degree* | "int(3, 9)" |

**Table 3.4:** Search space 2 definition.

### 3.5.5 Search space 3

Using the default value $\gamma_0$ seem to produce very consistent results in the sensitivity tests. Hence why it was not included in the search spaces earlier. This search space has 375 unique combinations on the conditional grid. Calculating all the combinations takes some considerable resources, but it

is still doable. For this reason it has been named the *big search space*. The point at which a search space becomes too big depends on the use case. For a detailed comparison analysis, each study may not bee to big. Because for statistical significance, each study has to be run multiple times on each dataset.

| HP | value |
|---|---|
| *kernel* | ["linear", "poly", "rbf"] |
| *scaling* | ["standard", "uniform", "unit"] |
| *cost* | "float(1e-2, 1e2, steps=5, log=True)" |
| *degree* | "int(3, 9)" |
| *gamma* | "float($\gamma_0$e-1, $\gamma_0$e1, steps=3, log=True)" [a] |

**Table 3.5:** Search space 3 definition. [a] if written in a study definition file, the limits have to be written as real numbers. In case of MNIST this would be "float(1.28e-4, 1.28e-2, log=True)".

### 3.5.6 Search space 4

This search space now includes HPs, which produced very consistent plateaus in the sensitivity tests. While this may seem to inflate the studies unnecessarily, recall the premise, that the user may not be aware of this observed consistency. Furthermore, the basis on which the search spaces were defined were significance tests. So there is a very good chance, that adjusting these HPs may give a combination, that could yield a model with good performance. With 5280 unique combinations on the conditional grid, computing all of them may exhaust available computing resources. This search space was mainly used to study, how a partial study could be performed. In this case, other stop criteria, besides the number of combinations are tested, which is discussed in section 4.4.

| HP | value |
|---|---|
| *kernel* | ["linear", "poly", "rbf"] |
| *scaling* | ["standard", "uniform", "unit"] |
| *cost* | "float(1e-2, 1e2, steps=5, log=True)" |
| *degree* | "int(3, 9)" |
| *gamma* | "float($\gamma_0$e-1, $\gamma_0$e1, steps=3, log=True)" |
| *coef0* | "float(0, 6, steps=4, log=True)" |
| *tol* | "float(1e-12, 1e-6, steps=4, log=True)" |

**Table 3.6:** Search space 4 definition.

## 3.6 Study definitions

Recall the definition of a study in section 2.4 each study samples HPs over its defined search spaces and evaluates the objective function until at least one specified stop criterion is triggered. For each combination of search spaces (section 3.5), datasets (section 2.2) and optimizers one study is

defined. As noted, the optimizers are Grid Search (conditional grid subsubsection 3.3.3), Random Search and TPE. Only the conditional Grid Search used the discretization of each search space. For the other search algorithms, the discretization was removed.

### 3.6.1 Full tests

The studies on search spaces 0 to 3 were run on their complete respective grids. Hence the name *full test*. The stop criterion was the number of combinations on the conditional grid. These are 9, 45, 135 and 375 for the search spaces 0, 1, 2 and 3 respectively, they were used for all sampler types. To obtain a statistically relevant results, each study was run five times.

### 3.6.2 Partial tests

As noted in subsection 3.5.6, search space 4 with 5280 combinations on the conditional grid is to big to be searched exhaustively. Assuming an average of two minutes per trial on the Deepsat dataset, this would result in a runtime of about seven days for a single study. Based on the results of the full tests, the stop criteria were changed, which is described in section 4.4.

## 3.7 Time trials

Another important quantity to set is the runtime limit of each dataset. As was established in subsection 2.5.3, there is a risk, that the PLSSVM model may never converge, or it may converge very slowly. However, it is important to keep in mind, that stopping any model early poses the risk of losing a well-performing model. Therefore a compromise has to be found. To measure this behavior, the simple HP search space 1 was used, to evaluate both runtime and errors. The runtime limits were set very high, to allow an analysis of the distribution of runtime per trial. With these findings we try to formulate a limit that represents a reasonable compromise between stopping non-converging models and not losing too many well-performing models. For this analysis, search space 1 with 45 combinations was chosen for three reasons:

- It is small enough, that calculation will finish in reasonable time.
- The number of trials is big enough, to get a coarse estimation of the runtime distribution.
- Due to the inclusion of the *cost* HP, the search should yield trials with varying runtime and performance. The importance of the *cost* HP can be seen in Figure 3.7b.

Search space 1 was run on a setup described in section 3.8.

To determine the timeout, the plots in Figure 3.8 were used. On the x-axis they show, how long a model took to fit. The bar graph is visualizes the distribution of models with different runtimes, which is the primary method of finding the timeout. Furthermore, the scatter plot shows the error on the validation dataset, described in section 2.2. The error refers to the missclassification error, described in subsection 2.1.1. This plot is used to determine the risk, of cutting of well-performing models.

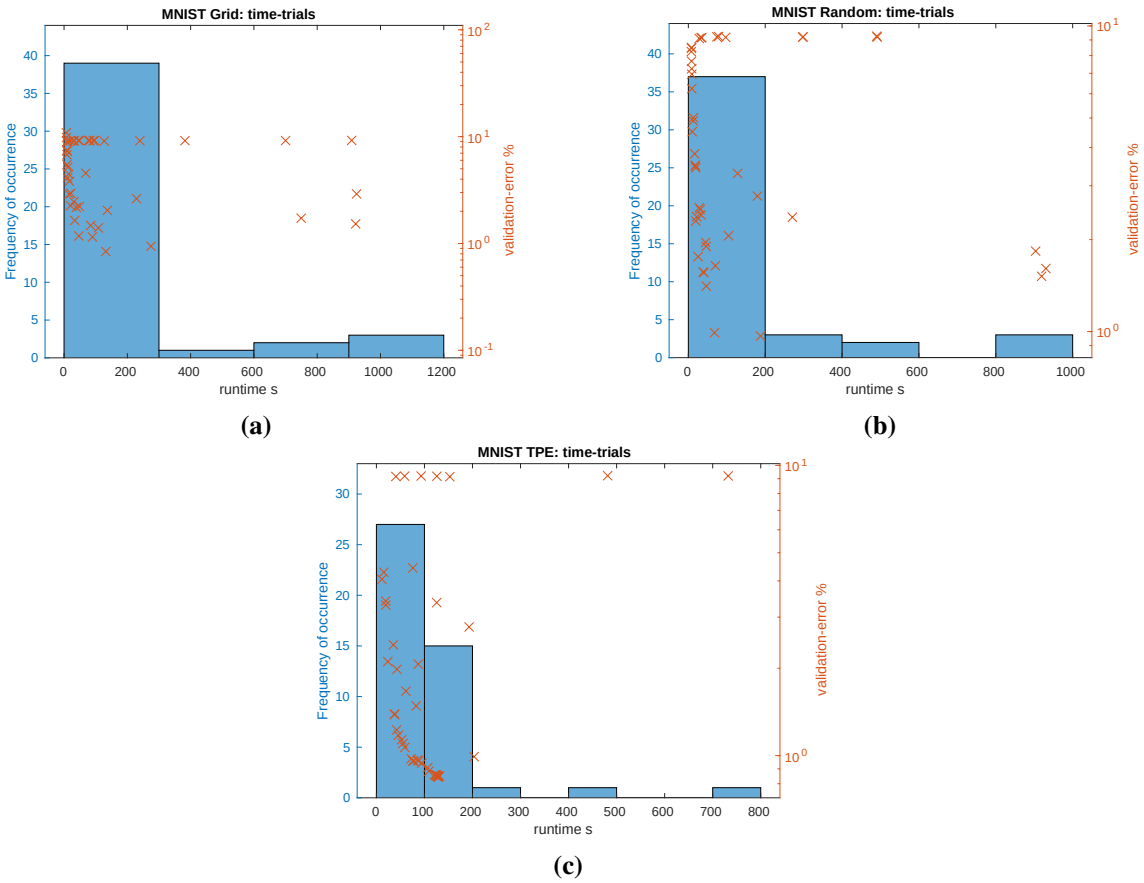



**Figure 3.8:** Different time tests on small search space. Time trials of the other datasets can be found in the Figures 1 and 2 of the appendix.

In all plots, there is no indication that suggests, that a longer fitting model performs significantly better. As such, the analysis is purely done using the bar graph. In each bar graph, only few models take any longer, than 300 s. Therefore, setting a timeout of 300 s would stop only few models. TPE seems to be a special case, where only very few models take longer than 200 s to fit. The surrogate model converged to an optimum of the cost parameter. As noted in section 3.4, the runtime is very sensitive towards the choice of cost parameter. So it is plausible, that a study converging to an optimum of this parameter may also converge to a certain runtime (in this case, 120 s). For this reason, the TPE time-trial shows fewer models, that take longer than 200 s. Yet from the other optimizers we know, that a reasonable number of models may take up to 300 s to fit. For this reason, the TPE optimizer may not be the best choice, to evaluate, how long a model will take to fit.

To sum up, a timeout of 300 s was determined for the MNIST dataset. Although to allow for some variance in computing time, the timeout was extended to a 330 s soft timeout and a 360 s hard timeout. Both types of timeouts are described in subsection 3.3.5. Using the same approach on the Software Defects and Deepsat Sat6 datasets, shown in Figures 1 and 2 of the appendix, which coincidentally arrived at the same timeouts.

## 3.8 Experiment settings

The purpose of this section is to provide insight into the experimental environment in order to make our results repeatable. The experiments were run on IPVS' *simcl1* cluster. It consists of two AMD EPYC 9274F server CPUs, each with 24 cores and 48 threads, it is equipped with 384GB of RAM. The scheduling software slurm [YJG03] divides simcl1 into four nodes, two of which have access to an NVidia A30 GPU with 24GB of VRAM each, and one node has access to an AMD Instinct MI210 with 64GB of VRAM. All tests were run on one of the NVidia GPU nodes with exclusive access, with PLSSVM running in the CUDA backend.

# 4 Results

This chapter presents and discusses the results of the studies as defined in section 3.5. Furthermore using these observations, optimized rules for searching large spaces are formulated.

## 4.1 GPU power draw

Recall the experiment setup (section 3.8), all studies were run on NVidia A30 GPUs and the energy consumption drawn by the GPU was logged. The first observation to establish is that in each study, the GPU power consumption was very consistent. Using Figure 4.1a as an example, most of the energy plots look comparable. For the small Software Defects dataset some minor deviations were observed, an example is Figure 4.2. The biggest outlier is still within 20% of the median (Figure 4.1b). Without doing any further research into the problem, it is hard to say, what causes these outliers. But it is likely one of several causes:

- Theoretically, certain HPs can lead to different complex models. For example, using a kernel other then linear will require transforming of the input samples as an extra step. It is not assumed, but it is possible, that a step like the kernel transformation may change the power consumption.
- The average power of a study is calculated by $\frac{\Delta E}{\Delta t}$, which is the energy consumed and runtime respectively.

  $$\Delta E = \int p(t)dt \tag{4.1}$$

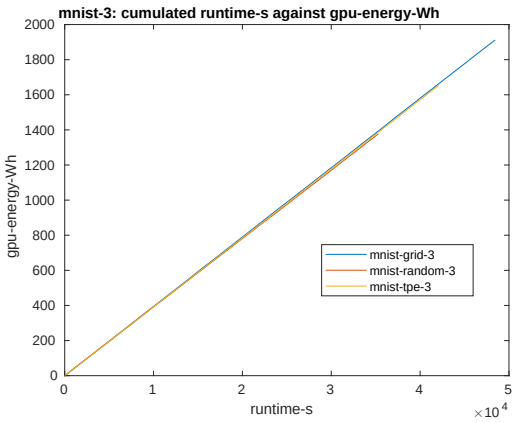
  the energy is calculated by taking the integral of the power over the runtime. Research suggests, that the NVidia-smi measurements deviate ±5% from the real value [YAA24]. At a rated Thermal Design Power (TDP)[1] of $165W$, this should be around $\pm 8.25W$ of electrical power draw. Additionally, the integration itself introduces a small error.
- While the experiments are running with exclusive slurm access to a node, the Operating System (OS) is still running tasks in the background. Furthermore, the separation of each node within slurm is not perfect and jobs on other nodes may interfere, especially when using shared resources, such as RAM or hard drives.

[1] TDP describes the dimensions of the cooling solution. While this is not necessarily the same as electrical power consumption, the average power at a continuous maximum computing load should still be at around this figure. This is due to the fact that almost all of a processor's electrical energy is converted to heat. The exact figure would depend on the computing load and the temperature of the processor.

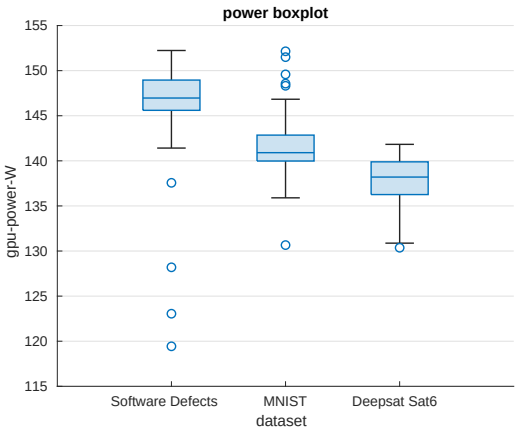
- The power consumption of any processor is a dynamic process. It is likely that for example during a short study, the power consumption of the GPU has not yet stabilized. This may be due to measurement errors (caused by averaging [YAA24]), as mentioned above, but it may also have a physical explanation: When a processor is idle, it cools down; when a load is present, it heats up. Because of the extensive cooling solution designed to dissipate a heat load of $165W$, the GPU takes some time to heat up. The efficiency of such a system is directly dependent on the temperature. A colder system requires less power to perform the same computations. In fact, the outlier in Figure 4.2 coincides with a short study. The search space 0 baseline studies are computed first, while the GPU may still be cold. Thus, this is a likely cause for this behavior.

However, we observe a quartile range of about $\pm 6W$ or $\pm 5\%$, which is consistent. Thus, we can assume, that for each of the selected datasets, runtime is interchangeable with the energy consumed.

Although it is important to note that the power drawn by the GPU during fitting seems to be dataset dependent (Figure 4.1b). It is likely, that the dataset structure (number of samples and number of features) is the factor behind this observation. In these measurements, the higher dimensional feature spaces (section 2.2) seem to result in lower power consumption. One explanation may be, that with a higher dimensional feature space, the GPU has to do more memory transfers during the process of fitting the dataset. The main computing units of any processor consumer the most power, but in this case they may be waiting for the data to be transferred. Nevertheless, this is an open avenue for further research. Especially due to the fact, that only one GPU from one vendor was studied. It is likely, that the qualitative behavior of other GPUs is different.




**(a)** Energy per runtime on one of the MNIST studies on search space 3.




**(b)** Each sample is the average power of a single study. These averages are plotted over the different datasets.

**Figure 4.1:** Within each dataset, the power of the conducted studies is very consistent.

While this does account of the energy consumed by the GPU, the CPU will consume energy at the same time. Measuring the energy was not part of this research, but should be addressed in future research. It is expected, that the energy consumed by the CPU will follow roughly the same linear pattern. It is possible, however, that the energy consumed by X86-type processors is harder to estimate due to their rich feature set.

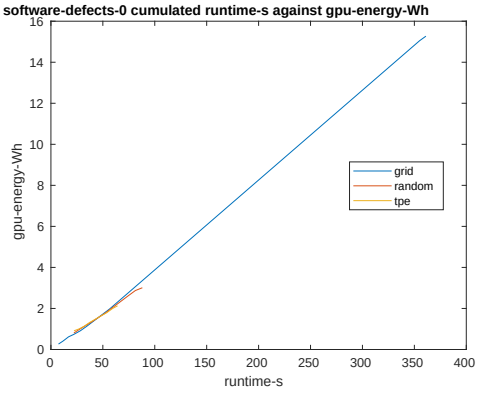



**Figure 4.2:** One of the power outlier on the Software Defects dataset using search space 0.

## 4.2 Performance results

This section discusses the final tuning results, without regard to efficiency, which is discussed in section 4.3. If an optimizer finds the model with the best score, as measured by accuracy on the validation holdout of the studies (section 2.4), it "wins". If the best model found by a study had a lower accuracy, it "looses". In case of search space 0, multiple optimizers may find the same optimum model and therefore multiple may win. This is summarized in Table 4.1.

Using the small search space 0, Grid Search is the best choice. This is to be expected, because it is a small and fully discrete space, only the Grid Search does guarantee, that the full grid was searched. For search space 1 and 2, TPE search seems the best choice. With three and four search axes, the Bayesian model is able, to find better optima outside the search grid. In the last search space 3, the Grid Search seems to win. This result is somewhat surprising due to the five search axes. Recall subsection 2.4.4, that Grid Search has a higher effective dimensionality. One explanation is, that Grid Search had an advantage due to it's exhaustive nature. It is feasible, that the Random Search sampled HPs too unevenly. Due to the dimensionality, it is possible, that the surrogate model of TPE may have gotten to complicated. This may have lead it to converge to a local optimum, instead of the global optimum. An explicit example of this behavior is discussed in the results of search space 2 subsection 4.3.4.

Furthermore, it is important to mention, that the choice of optimizer also depends on the dataset. While TPE was overall better on the search spaces 1 and 2, it is only on par with the Grid Search on the MNIST dataset. As is discussed in subsection 4.3.3, this may be related to the fact, that the Grid Search has an advantage, if the optimum is at the boundary of a search space.

| Search space | 0 | | 1 | | 2 | | 3 | | overall | |
|---|---|---|---|---|---|---|---|---|---|---|
| | win | loss | win | loss | win | loss | win | loss | win | loss |
| **overall** | | | | | | | | | | |
| Grid Search | **15** | **0** | 5 | 10 | 2 | 13 | 6 | 9 | 28 | 32 |
| Random Search | 11 | 4 | 2 | 13 | 3 | 12 | 5 | 10 | 21 | 39 |
| TPE | 12 | 3 | **9** | **6** | **10** | **5** | **9** | **6** | **40** | **20** |
| **Software Defects** | | | | | | | | | | |
| Grid Search | **5** | **0** | 0 | 5 | 0 | 5 | 0 | 5 | 5 | 15 |
| Random Search | 4 | 1 | 2 | 3 | 0 | 5 | **3** | **2** | 9 | 11 |
| TPE | **5** | **0** | **3** | **2** | **5** | 0 | 2 | 3 | **15** | **5** |
| **MNIST** | | | | | | | | | | |
| Grid Search | **5** | **0** | **5** | **0** | **2** | **3** | 3 | 2 | **15** | **5** |
| Random Search | 3 | 2 | 0 | 5 | 1 | 4 | 2 | 3 | 6 | 14 |
| TPE | 3 | 2 | 1 | 4 | **2** | **3** | **5** | **0** | 11 | 9 |
| **Deepsat Sat6** | | | | | | | | | | |
| Grid Search | **5** | **0** | 0 | 5 | 0 | 5 | **3** | **2** | 8 | 12 |
| Random Search | 3 | 2 | 0 | 5 | 2 | 3 | 0 | 5 | 5 | 15 |
| TPE | 4 | 1 | **5** | **0** | **3** | **2** | 2 | 3 | **14** | **6** |

**Table 4.1:** Win and loss comparison per search space and search algorithm and dataset.

## 4.3 In-depth search space comparison

This section will now discuss the efficiency of each algorithm. While the Table 4.1 may be an interesting starting point, it is more theoretical in nature. As will become apparent throughout this section, with the defined search spaces, all optimizers find a model with comparable performance. The performance difference between a winning or loosing model may be very small. However, other metrics, such as the performance variance or the runtime between the optimizers, can be very different. The search spaces are defined in section 3.5.

### 4.3.1 Default values

As a rough guideline, the default values defined in section 3.5 yield the following model performances:

| Dataset | Software Defects | MNIST | Deepsat Sat6 |
|---|---|---|---|
| $Acc_h$ [%] | 80.09 | 97.95 | 95.35 |
| $bias$ [%] | 0.18 | -0.21 | -0.16 |
| $t_{tr}$ [s] | 5.8 | 19.8 | 49.5 |

**Table 4.2:** Performance and runtime figures of one run using the default values.

Repeatedly fitting datasets with PLSSVM using these default values will always yield the same performance. Using the same experiment settings (section 3.8), the runtime will vary slightly. The accuracy was measured on the HPO holdout slice $D_{holdout}$ (section 2.4), the bias refers to the difference to the optimizer validation slice $D_{validation}$. A positive value indicates that the value was higher during model selection. $t_{tr}$ is the runtime a single trial. It measures the elapsed time of fitting the data on $D_{train}$ and predicting the data of $D_{train}$ and $D_{validation}$ to estimate performance.

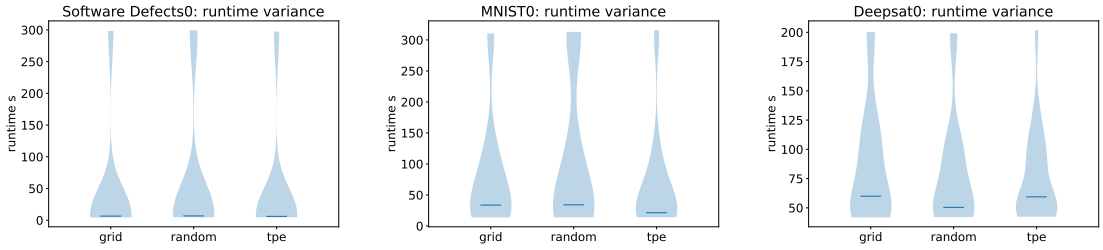
### 4.3.2 Search space 0

| Dataset | Software Defects | | | MNIST | | | Deepsat Sat6 | | |
|---|---|---|---|---|---|---|---|---|---|
| Tuner | Grid | Random | TPE | Grid | Random | TPE | Grid | Random | TPE |
| win | **5** | 4 | **5** | **5** | 3 | 3 | **5** | 3 | 4 |
| loss | **0** | 1 | **0** | **0** | 2 | 2 | **0** | 2 | 1 |
| $\overline{Acc_h}$ [%] | **81.132** | 80.985 | **81.132** | **98.330** | 98.218 | 98.198 | **97.384** | 97.366 | **97.384** |
| std | 0 | 0.3296 | 0 | 0 | 0.1534 | 0.1842 | 0 | 0.1090 | 0.08765 |
| $\overline{bias}$ [%] | 0.319 | 0.277 | 0.319 | 0.028 | 0.107 | 0.020 | -0.131 | -0.174 | -0.123 |
| std | 0 | 0.0935 | 0 | 0 | 0.108 | 0.154 | 0 | 0.121 | 0.0188 |
| $\bar{t}_{st}$ [s] | 360 | 480 | **359** | 698 | 933 | **580** | 766 | 703 | **671** |
| min | 352 | 88 | 64 | 360 | 589 | 234 | 760 | 606 | 615 |
| max | 364 | 957 | 652 | 705 | 1072 | 731 | 773 | 848 | 724 |
| $\bar{t}_{tr}$ | 40.0 | 53.3 | **39.9** | 77.5 | 103 | **64.4** | 85.1 | 78.1 | **74.6** |
| best HPs | Grid @81.132% sc=standard, k=rbf | | | Grid @98.830% sc=uniform, k=poly | | | Grid @97.384% sc=standard, k=rbf | | |

**Table 4.3:** Performance and runtime comparison of search space 0. $\bar{t}_{st}$ is the average runtime of a complete study, while $\bar{t}_{tr}$ is the average runtime of a single trial. *Best* describes, which optimizer found the model with the highest accuracy on the study holdout in all experiments.

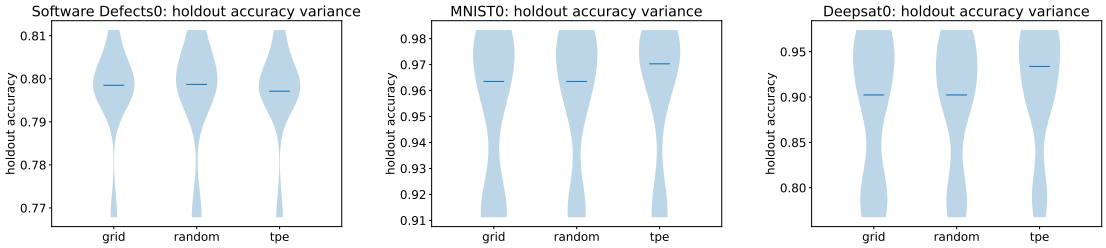
On search space 0, the difference between each optimizer should be minimal. Indeed, in case of model performance, this is the case. Due to it's exhaustive nature, the Grid Search always finds the optimum within the grid. Furthermore, it always requires about the same time to finish, within a narrow window. This is not the case for the other optimizers. While their results are close and they almost always found the same optimum on the grid, their runtime varied considerably. The runtime itself is dependent on the HPs. In case of the TPE and Random Search, both are not grid based and both may sample previously sampled combinations. Due to this, there is a small chance, that the optimum was not within all the sampled HPs. With only nine evaluations, TPE is still doing Random Search, as explained in subsection 2.4.4. This is also the reason for the runtime differences: The longer running studies (especially on Software Defects) sampled a long running HP combination multiple times, whereas the shorter running combinations may have missed the same combination. For the Software Defect dataset, one combination {kernel (k)=poly, scaling HP (sc)=standard} on the grid does not converge before the soft timeout (subsection 3.3.5). Fitting this combination takes much longer, then any other combination, which can be seen in Figure 3b. In fact, it is possible, this combination does not converge. The variance of runtime per study is less on the other datasets, it is shown in Figure 4.3. These graphs compare the runtime variance per study,

it shows the distribution of runtime per trial. In particular Figure 4.3 shows, that only very few models take around 300 s to fit on Software Defects. So any study, that fits this model multiple times will take much longer and any study, that misses this model will be done much quicker.




**(a)** **(b)** **(c)**

**Figure 4.3:** Search space 0 runtime distributions. The line denotes the median of each distribution, the number besides the dataset name indicates the search space.




**(a)** **(b)** **(c)**

**Figure 4.4:** Search space 0 score distributions over the trials of all studies combined.

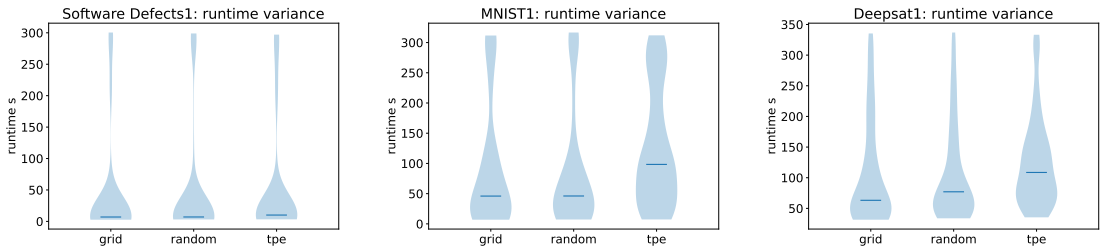
### 4.3.3 Search space 1

On search space 1, the cost (C) search axis was added. This axis is continuous and as such, this allows the continuous samplers to work more efficiently compared to a grid, as described in subsection 2.4.4. As a result, on two of three datasets, the continuous samplers find the best models. On MNIST, many of the better models have higher sampled cost values. In case of Grid Search, the sampled cost value was exactly at the boundary of the search axis. Therefore it is likely, that the Grid Search had an advantage on MNIST, as it does sample all values on the boundary of a search space. Continuous samplers are less likely to sample values on the boundary of a search axis, as can be seen in Figure 2.6. Furthermore, it is likely, that increasing the cost HP may increase the performance further. Yet it is important to keep in mind, that this would also result in exponentially higher time to fit the model. In the worst case, a model may never converge. This is discussed in section 3.4. One other likely explanation would be, that using TPE the surrogate has trouble exploring the boundary of a search space. This is due to the Gaussian Process: Each HP starts with a normal distributed prior. Meaning, that each HP is more likely to be sampled from the center of their defined search axis.

| Dataset | Software Defects | | | MNIST | | | Deepsat Sat6 | | |
|---|---|---|---|---|---|---|---|---|---|
| Tuner | Grid | Random | TPE | Grid | Random | TPE | Grid | Random | TPE |
| win | 0 | **3** | 2 | **5** | 0 | 1 | 0 | 0 | **5** |
| loss | 5 | **2** | 3 | **0** | 5 | 4 | 5 | 5 | **0** |
| $\overline{Acc_h}$ [%] | 81.132 | **81.213** | 81.207 | **98.870** | 98.790 | 98.828 | 97.999 | 98.011 | **98.029** |
| std | 0 | 0.05259 | 0.03680 | 0 | 0.04637 | 0.03033 | 0 | 0.01919 | 0.00626 |
| $\overline{bias}$ [%] | 0.32 | 0.28 | 0.28 | 0.29 | 0.28 | 0.267 | -0.042 | -0.040 | -0.034 |
| std | 0 | 0.017 | 0.037 | 0 | 0.012 | 0.029 | 0 | 0.015 | 0.0063 |
| $\bar{t}_{st}$ [s] | 1978 | **480** | 2322 | 4216 | **4024** | 5724 | 5070 | **4840** | 5634 |
| min | 1954 | 88 | 744 | 4196 | 3682 | 3806 | 4867 | 4233 | 5373 |
| max | 1996 | 957 | 4414 | 4247 | 4332 | 6827 | 5807 | 5398 | 5930 |
| $\bar{t}_{tr}$ | 44.0 | **10.7** | 51.6 | 93.7 | **89.4** | 127 | 113 | **108** | 125 |
| best HPs | Random @81.245% sc=standard, k=rbf, C=0.366 | | | Grid @98.870% sc=uniform, k=poly, C=100 | | | TPE @98.038% sc=standard, k=rbf, C=19.0 | | |

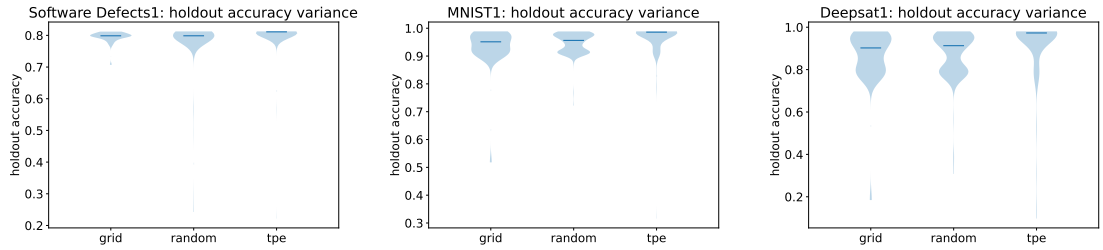
**Table 4.4:** Performance and runtime comparison of search space 1.

Again, the highest performance increase can be observed on the Deepsat Sat6 dataset. The best model had an accuracy increase of 0.65%. Comparing the continuous samplers on Software Defects and Deepsat, there is a correlation between bias and optimizer on this search space. With the lower absolute bias of Deepsat, the models found by the TPE sampler are better on the study validation holdout. This may be the reason, why TPE found models with higher performance. A reduced bias means, that the real performance of the selected model is closer to the one known by the optimizer. This may be achieved by increasing the size of the validation dataset, although this is likely to reduce the performance of the found model as well. Theoretically one may also try using CV instead. As described in section 2.2, this is likely to reduce the pessimistic bias of the simple holdout approach used in these experiments. While this will drastically increase the required resources, due to the low number of trials on this search space, a 5 or 10 fold cross validation would still be feasible.




**(a)** **(b)** **(c)**

**Figure 4.5:** Search space 1 runtime distributions.




**(a)** **(b)** **(c)**

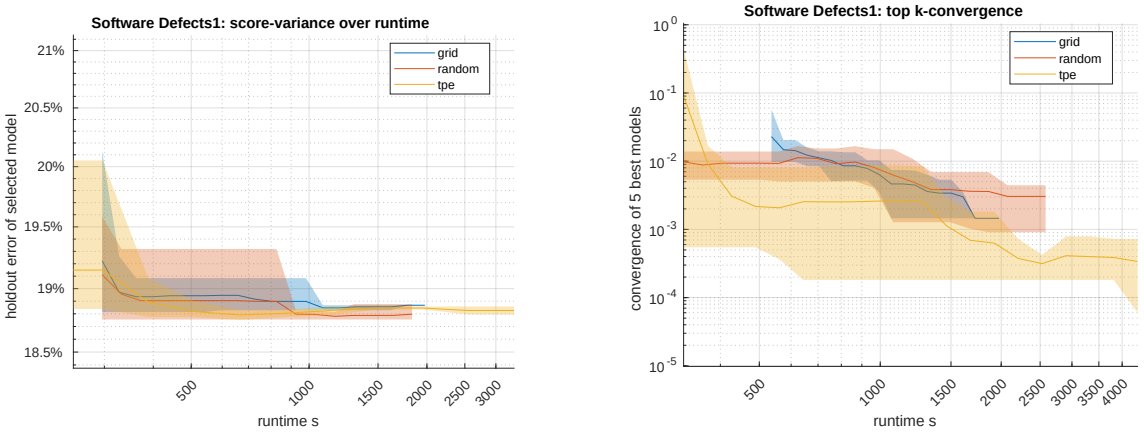
**Figure 4.6:** Search space 1 score distributions.

In all cases, the models found by the TPE sampler have a smaller standard deviation in their model performance. On the Deepsat Sat6 dataset all five studies of TPE found very similar HPs. Scaling and kernel were always identical, the average of the cost HP was 16.5, whereas its standard deviation was 1.88. All TPE studies found the same optimum, so it is likely, they all found the global optimum within the defined search space. Another trend that can be observed from here on (Figure 4.6) is that the median score of each study within the TPE studies is always close to its respective optimum score.

The accuracy on the study validation holdout is plotted over the runtime in Figures 4.7a, 4.8a and 4.9a. Instead of using accuracy directly, it was transformed into the *missclassification error* (subsection 2.1.1). This allows visualization on a logarithmic scale, smaller values are further apart and therefore more visible. The line of each plot is the mean of all five conducted studies at the sampled runtime. The upper and lower edges are the maximum and minimum error of the five studies, respectively.In most cases, the optimizer found a good model before executing the last trial. In case of Figure 4.7a, we may stop tuning after 20 min and the model performance should still be very good. One limitation of this type of plot is, that it only depicts the best and worst model at that point in time. If those two are far apart, this does not mean, both optimizer have not converged. It is possible, as is discussed in subsection 4.3.4, that two studies converge towards different optima. To visualize the convergence behavior, the relative difference of the top-5 holdout validations are plotted. This is calculated as

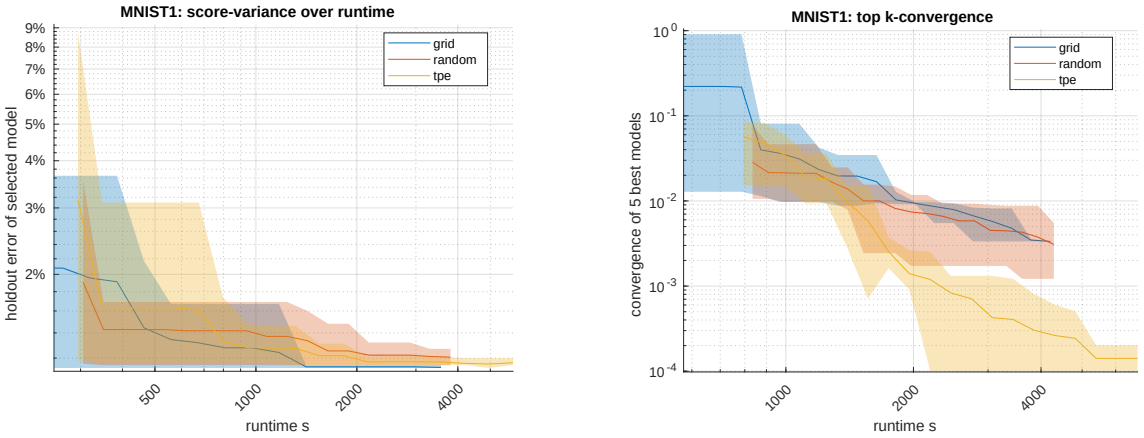
$$\frac{Acc_1 - Acc_5}{Acc_5}, \tag{4.2}$$

where $Acc_1$ denotes the highest accuracy of all found models so far and $Acc_5$ the fifth highest. Using the top-5 convergence gives some insight, how much a optimizer has converged so far. The lower the score, the smaller the change and therefore models found in further trials are expected to not to be much better. In all cases, the TPE optimizer has a lower top-5 convergence difference on average. This is due to the fact, that TPE exploits optima. Many sampled HPs combinations are near a found optimum, so the score converges. Basing a stopping criterion around this behavior may not yield the best performing model, but it it very likely to find a well performing model in shorter time.




**(a)** **(b)**

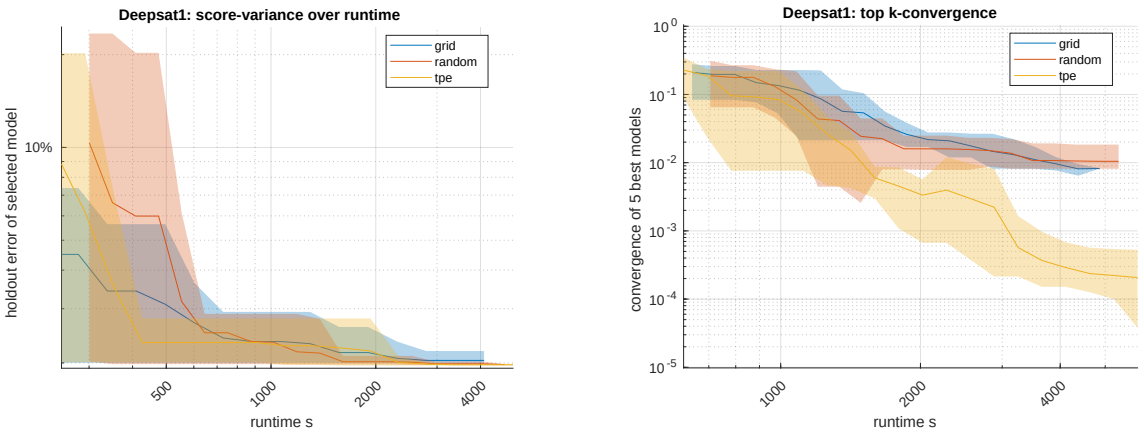
**Figure 4.7:** Search space 1 Software Defects score and convergence.




**(a)**

**(b)** TPE studies had a top-5 difference of zero, which which was cut off on the log-plot.

**Figure 4.8:** Search space 1 MNIST score and convergence.




**(a)** **(b)**

**Figure 4.9:** Search space 1 Deepsat Sat6 score and convergence.

### 4.3.4 Search space 2

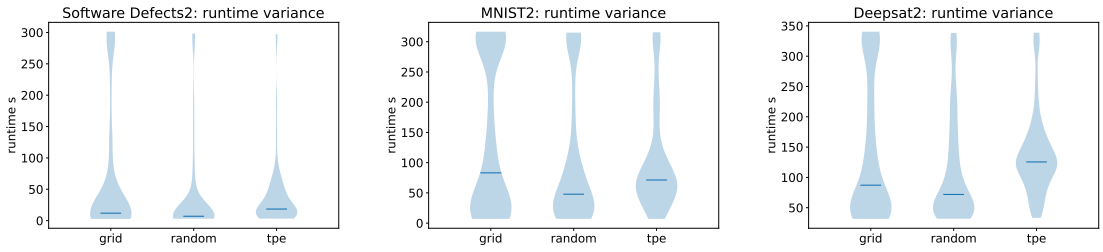
On this search space, the polynomial degree (d) search axis is introduced. This only yielded minor improvement, the biggest one was 0.1% of the best model found on the MNIST dataset. In this case, the continuous samplers were on par with the Grid Search on MNIST. Given, that this axis is again discrete, it is somewhat surprising, that the continuous samplers improved. It is a possible consequence of the fact, that using a polynomial kernel, the matching cost HP is not at the boundary of its search space. Another consequence of introducing the new search axis is, that one of the TPE studies on Deepsat got stuck in a local optimum. This study found a optimum at the ninth degree polynomial kernel. It diverged from all other four studies, that found the optimum at the rbf kernel, which was already described in subsection 4.3.3. The other studies on the Deepsat Sat6 dataset found the same optimum and as such, the new search axis does not benefit the tuning for this dataset. This can also be seen in the Figures 4.14a and 4.14b. According to the second plot, all TPE studies seem to have converged well, while their score variance in the first plot is still very high.

And while it reduced the average performance of TPE, it did boost that of Random Search. Random Search works by sampling the unconditional kernel HP first (see subsection 3.3.3). The number of each drawn linear, polynomial or rbf kernel HP is expected to be a third of all trials. Next the conditional polynomial HPs, like degree are drawn. While the expected value of sampled rbf kernels on search space 1 is 15, for the search space 2 this would be 45. Due to the fact, that the degree HP remains unused for the rbf kernel, all 45 trials are only evaluated on the scaling and cost HPs (as was on search space 1). However, it is likely that the expected number of 45 trials of the polynomial kernel may be a disadvantage for a case, where the optimum is within the polynomial kernel. This imbalance of the Random Search opens the possibility for further research.

Another observation is, that using this search space introduced many longer running trials within the conditional grid sampler. This is depicted in Figure 4.10, it is particularly visible for the MNIST dataset. It is likely caused by the fact, that the introducing a HP dependent on the polynomial kernel increased the size of the polynomial subspace of the conditional grid, compared to the other subspaces, that remain constant. Combinations of polynomial kernel with standard scaling are likely to converge only slowly on the MNIST dataset as depicted in Figure 3.7a. Conversely, TPE

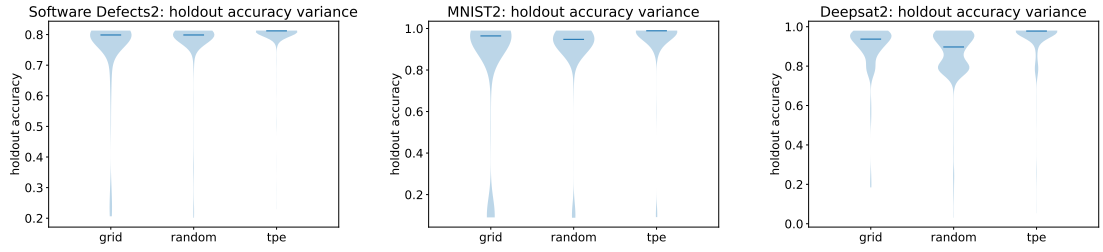
| Dataset | Software Defects | | | MNIST | | | Deepsat Sat6 | | |
|---|---|---|---|---|---|---|---|---|---|
| Tuner | Grid | Random | TPE | Grid | Random | TPE | Grid | Random | TPE |
| win | 0 | 0 | **5** | **2** | 1 | **2** | 0 | 2 | **3** |
| loss | 5 | 5 | **0** | **3** | 4 | **3** | 8 | 3 | **2** |
| $\overline{Acc_h}$ [%] | 81.132 | 81.223 | **81.253** | **98.930** | 98.912 | 98.928 | 97.999 | **98.031** | 97.985 |
| std | 0 | 0.03358 | 0.00828 | 0 | 0.02588 | 0.03114 | 0 | 0.04868 | 0.1094 |
| $\overline{bias}$ [%] | 0.319 | 0.271 | 0.184 | 0.262 | 0.266 | 0.269 | -0.042 | -0.041 | -0.027 |
| std | 0 | 0.0180 | 0.0898 | 0 | 0.142 | 0.0171 | 0 | 0.0199 | 0.0163 |
| $\bar{t}_{st}$ [s] | 8962 | **4826** | 4992 | 18841 | 13834 | **13830** | 19147 | **15162** | 18703 |
| min | 8911 | 3206 | 4366 | 18803 | 12986 | 9286 | 19062 | 13665 | 17870 |
| max | 9038 | 5757 | 6493 | 18872 | 15335 | 24587 | 19207 | 17215 | 19745 |
| $\bar{t}_{tr}$ | 66.4 | **35.7** | 37.0 | 140 | 102 | **102** | 142 | **112** | 139 |
| best HPs | TPE @81.265% sc=uniform, k=poly, d=9, C=4.05 | | | TPE @98.970% sc=uniform, k=poly, d=9, C=0.127 | | | TPE @98.041% sc=standard, k=rbf, C=18.8 | | |

**Table 4.5:** Performance and runtime comparison of search space 2.




**(a)** **(b)** **(c)**

**Figure 4.10:** Search space 2 runtime distributions.




**(a)** **(b)** **(c)**

**Figure 4.11:** Search space 2 score distributions.

sampled HP combinations, such that the runtime distributions concentrate around the median. The

combinations near the optimum all have a comparable runtime per trial. It is likely, that this has occurred on search space 1, but to a lesser degree. Since TPE can exploit optima more efficiently with 90 more trials compared to search space 1[2].

Although the validation error should decrease over time, in certain cases it may increase. This behavior is clearly visible in Figure 4.12a, where the validation error of the models selected by Grid Search, increases after 30min. The cause of this behavior is bias. Recall, the optimizer only optimizes given the accuracy on the $D_{validation}$ dataset within $D_{optimizer}$ (subsection 2.4.1). It does not know about the accuracy on the studies $D_{holdout}$, which is measured here. Selecting a model with higher accuracy on $D_{validation}$ often, but not always translates into higher accuracy on $D_{holdout}$. This behavior is repeated in Figure 4.13a, where the error of both Randoms Search and TPE increases after 1500s.

As on search space 1, on this search space, one may formulate criteria to stop early as well. For both MNIST and Software Defects, one can stop at about half an hour without dramatically affecting the model found. In both cases TPE seems to be better, if the study was stopped then. The difference, of their average scores may not be that different, but the maximum error of TPE in Figure 4.13a is lower, then that of the other optimizers. This does coincide with Figure 4.13b, where beyond this point, the convergence difference of TPE seems to stagnate. However, defining such an early stop is not so clear for Deepsat, which converges more gradually. One may choose 4000s, in which case the Random Search seems to find the best models. In this case, the difference between all Random Search studies is rather low, at around 0.1% accuracy difference.

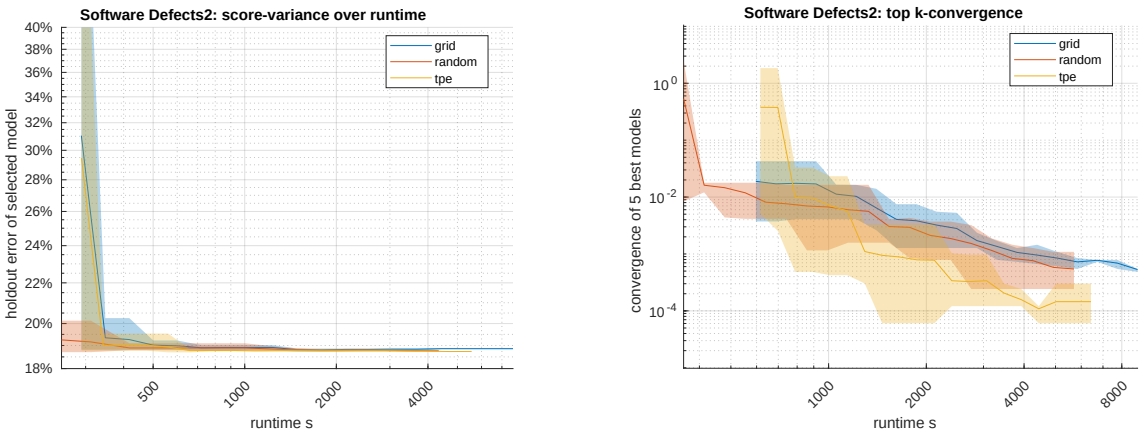
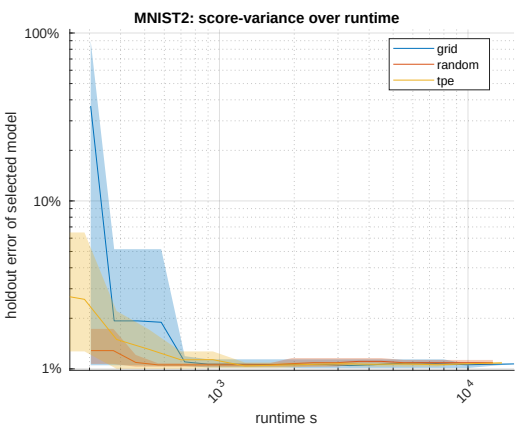
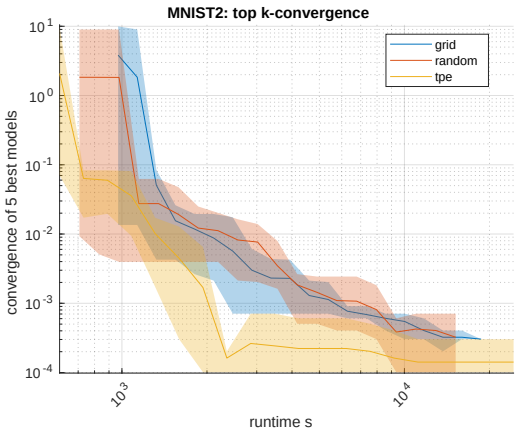



**Figure 4.12:** Search space 2 Software Defects score and convergence.

[2]According to its documentation (https://optuna.readthedocs.io/en/stable/reference/samplers/index.html [checked 11.04.24]), Optuna recommends to use TPE with around 100 to 1000 trials.

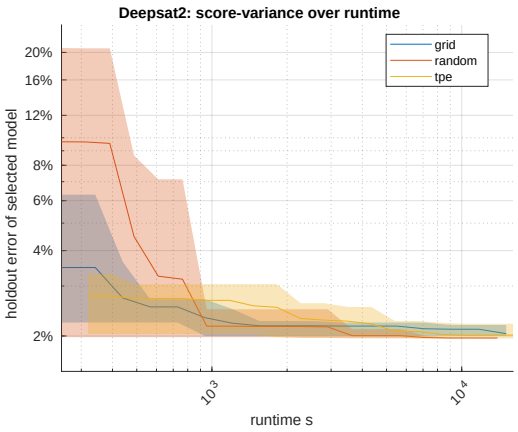



(a)

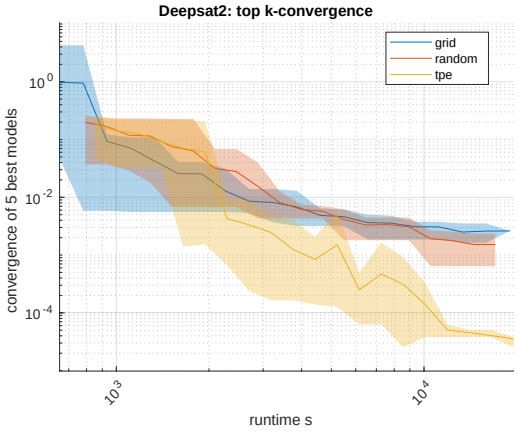



**(b)** TPE studies had a top-5 difference of zero, which which was cut off on the log-plot.

**Figure 4.13:** Search space 2 MNIST score and convergence.




(a)




(b)

**Figure 4.14:** Search space 2 Deepsat Sat6 score and convergence.

### 4.3.5 Search space 3

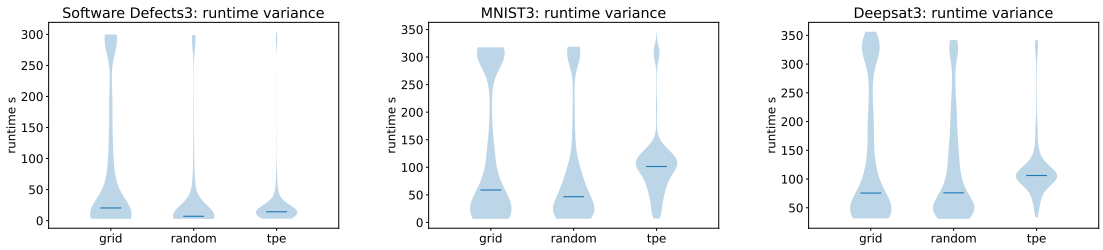
With search space 3, the $\gamma$ search axis is added. Like the cost search axis, it is continuous for the continuous samplers. Interestingly, on Software Defects the average performance all found models decreased due to increased bias. This may be a likely reason, why the performance TPE is on par with Random Search, even though TPE performed better on search space 2. It is likely, that tuning both $\gamma$ and cost leads to slight overfitting of the $D_{validation}$ dataset within $D_{optimizer}$.

Comparing the performance of search space 3 models, they are on par with models in search space 2. The average increase in performance is very small. However, it is noteworthy that on Figure 4.18a always found a optimum way quicker, then any other optimizer on this dataset. Even though its final performance may not be as good, as the one of the exhaustive Grid Search, it seems to be very resource efficient in this experiment. One possible reason, why TPE has trouble finding the optimum may be, that models on MNIST are fairly robust: As was observed, the performance

| Dataset | Software Defects | | | MNIST | | | Deepsat Sat6 | | |
|---|---|---|---|---|---|---|---|---|---|
| Tuner | Grid | Random | TPE | Grid | Random | TPE | Grid | Random | TPE |
| win | 0 | **3** | 2 | **3** | 2 | 0 | **3** | 0 | 2 |
| loss | 5 | **2** | 3 | **2** | 3 | 5 | **2** | 5 | 3 |
| $\overline{Acc_h}$ [%] | 81.127 | 81.219 | **81.237** | **98.960** | 98.938 | 98.930 | 98.199 | 93.139 | **98.199** |
| std | 0 | 0.07870 | 0.01966 | 0 | 0.05541 | 0.01414 | 0 | 0.05425 | 0.00507 |
| $\overline{bias}$ [%] | 0.330 | 0.265 | 0.277 | 0.240 | 0.254 | 0.272 | -0.076 | -0.018 | 0.0276 |
| std | 0 | 0.0753 | 0.0860 | 0 | 0.0466 | 0.0202 | 0 | 0.05097 | 0.0631 |
| $\bar{t}_{st}$ [s] | 30966 | 13898 | **9706** | 48443 | **34509** | 39618 | 51370 | 44339 | **44274** |
| min | 30839 | 12102 | 7470 | 48321 | 28916 | 28471 | 50993 | 43457 | 42721 |
| max | 31114 | 15422 | 12911 | 48667 | 37235 | 44715 | 51569 | 47401 | 46105 |
| $\bar{t}_{tr}$ | 82.6 | 37.1 | **25.9** | 129 | **92.0** | 106 | 137 | 118 | **118** |
| best HPs | Random @81.294% sc=standard, k=rbf, C=0.072, $\gamma = 1.81\gamma_0$ | | | Random @99.01% sc=uniform, k=poly, d=8, C=0.0155, $\gamma = 1.79\gamma_0$ | | | TPE @98.207% sc=standard, k=rbf, C=9.86, $\gamma = 2.94\gamma_0$ | | |

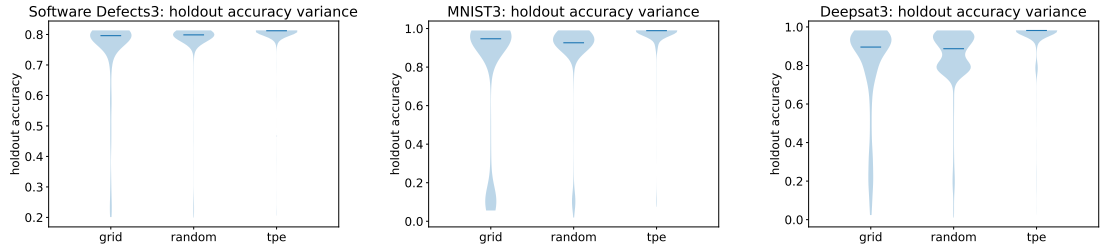
**Table 4.6:** Performance and runtime comparison of search space 3, $\gamma_0 = 1/n_{features}$.

increase on MNIST over the last search spaces was small. If many HP combinations produce a good model, then the difference between each model is minute. In this case the surrogate model may have trouble suggesting new samples, that will improve its performance.




**(a)** **(b)** **(c)**

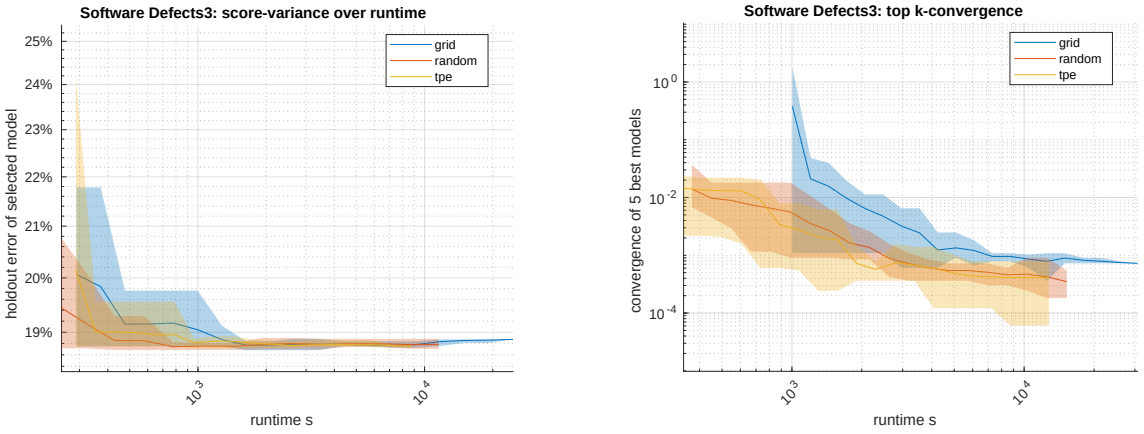
**Figure 4.15:** Search space 3 runtime distributions.




**(a)** **(b)** **(c)**

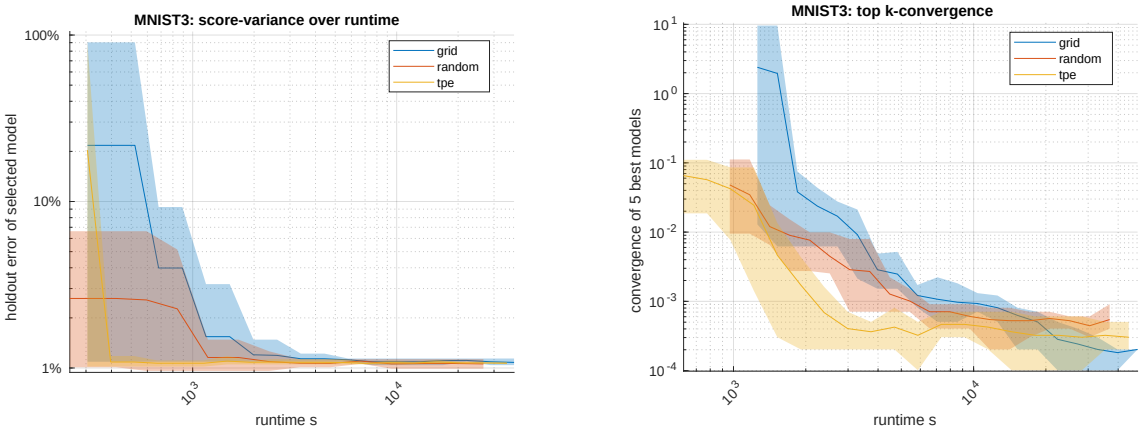
**Figure 4.16:** Search space 3 score distributions.

Conversely, the new search axis boosted the performance of Deepsat Sat6 models by about 0.2% on average. In this search space, TPE and Grid Search converge to a similar performing model. Yet TPE got stuck in local optima more often. The best study, converged towards an optimum at $2.7\gamma_0$ and the resulting model had an accuracy of 98.207%. Whereas the other TPE studies converged to an optimum at about $10\gamma_0$, which coincides with the one found by Grid Search. Without more insight into the surrogate model, it is not trivial to speculate, what went wrong. However, it is is possible that the new search axis has increased the complexity of the surrogate model so that it no works as efficiently. Another possibility is, that the optimum at $2.7\gamma_0$ is smaller. This means that only small deviations in $\gamma$ (and all other HPs of the search space) may lead to a bigger changes in the score. In this case, this optimum would be harder to find. Again, like on the MNIST dataset, the TPE search converges quicker towards the optimum, then the other optimizers. On all three datasets, one can may implement an early stopping criterion. Except for Grid Search on Deepsat Sat6, an early stopping is not likely, to change the final result much.



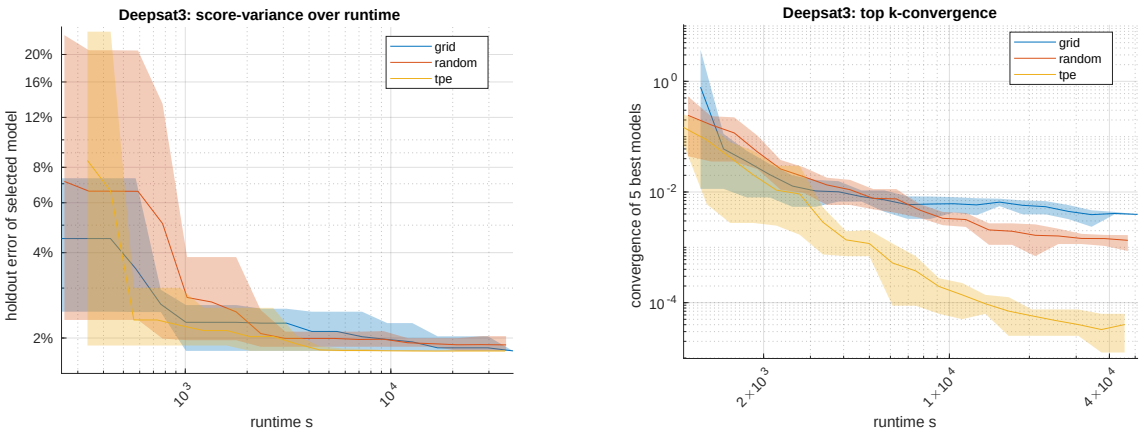

**(a)** **(b)**

**Figure 4.17:** Search space 3 Software Defects score and convergence.




**(a)** **(b)**

**Figure 4.18:** Search space 3 MNIST score and convergence.




**(a)** **(b)**

**Figure 4.19:** Search space 3 Deepsat Sat6 score and convergence.

## 4.4 Time Optimized Stop Criteria

Due to the fact, that SVM seems to be quite robust on the selected datasets, an exhaustive search may not be necessary, to find a good model. As noted in several search space discussions in section 4.3, terminating a study early may still yield very good results while reducing the required computing resources. Two relevant criteria were identified:

- A time budget, where a study runs trials, until a time limit is reached. One difficulty would be, that the user does not know, how long it would take, until a optimizer is likely to have found a well performing model.
- Alternatively, one can define a criterion based on the convergence of the optimizer. If it appears, that the performance improvements are small, then a study is stopped.

The second approach, called the *convergence stop* works like the top-k diagrams. Although in the context of an optimizer, instead of the accuracy on the study validation holdout, this criterion uses the SVMScore subsection 3.3.4, which usually is the accuracy of the $D_{validation}$ dataset slice within $D_{optimizer}$. The k best SVMScores are saved within a vector $\sigma$. If their relative difference is less, then a set threshold, then the study is halted.

$$\frac{max(\sigma) - min(\sigma)}{min(\sigma)} < \tau_c \tag{4.3}$$

The theory behind this approach is, with the observed combinations of objective functions and datasets, multiple trials of each optimizer come close to a performance limit. This is similar to top-5 convergence plots, one example is section 4.3. As such, the min and max of the $k$ best models should converge after sufficient trials. Again, just as with the simple score threshold, this bears the risk, that it may not work as single stop criterion. There are only so many models, that reach a certain performance. So if $k$ is too high, $\sigma$ never converges below $\tau_c$. For this reason, another constrain, such as runtime or number of trials must be set.

A related criterion was developed in [RDPCV+18, cpt. IV], although they did not use a constant k, but rather a relative number. They formulated a criterion that would stop, when the variance of the best 25% models is less then 0.01. This has the advantage, that it may be simpler to set for the

user. Although due to the dynamic nature of this criterion, it will compare an increasing number of models. For an optimizer, that has a high median score, compared to the best score, this may stop too early. But for optimizer with a bad median score (as for example Grid Search in Figure 4.26b), this criterion may never stop. This is, because the chance of a good model is lower, so the expected relative number of good models compared is low as well. In fact, in their work, all but one study stopped due to a timeout instead. Further research into the topic would be required, to verify this hypothesis.

Both criteria were set using the results found in section 4.3. The time limits were set to 30 min for Software Defects, 60 min for MNIST and 90 min for Deepsat Sat6. For the convergence stop, $10^{-2}$ was chosen as the relative threshold. This number was chosen because once this value is reached in the top 5 convergence plots, the results shown in the corresponding plots are close to the optimum. To complement the convergence stop, a time limit was set to double the time limit defined for the timed searches.

To find out, if optimizing for runtime helps, a fourth study was setup. Called the *TimeTPE* study, it uses TPE as optimizer and the *time metric* described in subsection 3.3.4. TPE will optimize primarily for model performance, but as a secondary objective it reduces the runtime of each trial. All three samplers and the TimeTPE study were run on every dataset, optimizing search space 4 (section 3.5) using both stop criteria described in this section.

### 4.4.1 Timebound search

Using a limited time seems to benefit the TPE sampler. It had the best average results on all three datasets. It is very likely, that with the aid of the surrogate, TPE can converge towards a optimum quicker, which can be observed in Figures 4.22a and 4.24a. However, on the MNIST dataset, it is just on par with the Random Search, probably for the same reasons that TPE performed on par with the grid search in subsection 4.3.3. Yet, TPE still converges to its optimum faster, which can be observed in Figure 4.23. It is important to note, that in all cases, if Grid Search and Random Search were run longer, then they may still find a better optimum, as was the case with the fulltest studies in subsection 4.3.5. One possible reason, why the Grid Search performs worse is the poor accuracy variance. This can be seen in Figure 4.21, particularly on MNIST, or Deepsat, where many of the combinations on the Grid have a very bad accuracy, compared to the all the other search spaces. It is possible, that due to the new search axes, the HP combinations at the boundary or in the corner of the new search space are particularly bad.

Compared to any other optimizer, the TimeTPE search greatly reduces the time per trial, as can be seen in Figure 4.20. Due to the reduced time per trial, the TimeTPE study is able to test more configurations in the same time. However, due to its secondary goal, accuracy is reduced in all tests. In both MNIST and Deepsat Sat6, its accuracy is improved, compared to the default values subsection 4.3.1, while it even decreased on the Software Defects dataset. Recall the correlation between score and cost parameter in Figure 3.7b, in this plot, another correlation was determined for the polynomial and rbf kernel: A higher cost parameter can result in a higher score, yet it also increases the fitting time. Thus, reducing the cost can increase the multi-objective formulated by the time-metric, yet the performance of the model is reduced. It is important to mention however, that not all models found by TimeTPE have a much worse performance, as can be seen in Figure 4.21. Therefore, it might be an interesting to research, whether a split metric approach may help in this

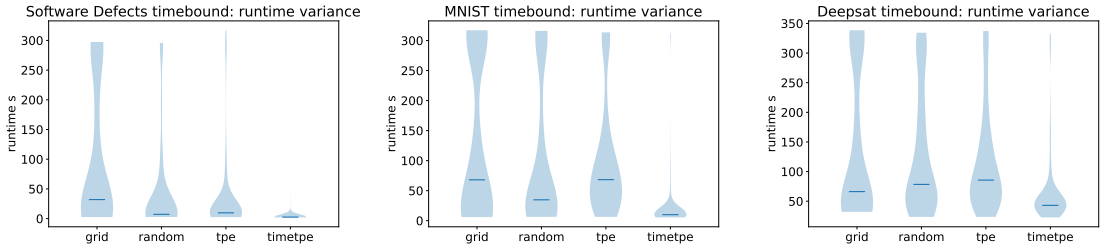
| Dataset | Software Defects | | | | MNIST | | | |
|---|---|---|---|---|---|---|---|---|
| Tuner | Grid | Random | TPE | TimeTPE | Grid | Random | TPE | TimeTPE |
| win | 1 | 1 | **3** | 0 | 1 | **2** | **2** | 0 |
| loss | 4 | 4 | **2** | 5 | 4 | **3** | **3** | 5 |
| $\overline{Acc_h}$ [%] | 81.151 | 81.186 | **81.249** | 78.410 | 98.904 | 98.908 | **98.918** | 98.214 |
| std | 0.202 | 0.0973 | **0.0860** | 1.44 | 0.0971 | **0.0811** | 0.1823 | 0.708 |
| $\overline{bias}$ [%] | 0.101 | 0.226 | 0.207 | 0.224 | 0.221 | 0.211 | 0.232 | 0.0774 |
| std | 0.0439 | 0.0436 | 0.162 | 0.158 | 0.0791 | 0.0541 | 0.0570 | 0.196 |
| $\bar{t}_{st}$ [s] | 1811 | 1858 | 1861 | 1632 | 3656 | 3658 | 3630 | 3319 |
| min | 1789 | 1758 | 1791 | 1552 | 3560 | 3568 | 3538 | 3301 |
| max | 1838 | 1984 | 1993 | 1688 | 3773 | 3826 | 3840 | 3335 |
| $\bar{t}_{tr}$ | 96 | 37.1 | 38.1 | **5.4** | 134 | 104 | 120 | **23.6** |
| best HPs | TPE @81.373% sc=unit, k=poly, d=8, coef0=3.60, C=4.63, $\gamma$=5.42$\gamma_0$, $\epsilon$=6.58e-10 | | | | Grid @99.07% sc=uniform, k=poly, d=8, coef0=0, C=100, $\gamma = \gamma_0$, $\epsilon$=1e-6 | | | |

**Table 4.7:** Performance and runtime comparison table A of search space 4 using a timebound stop criterion.

| Dataset | Deepsat Sat6 | | | |
|---|---|---|---|---|
| Tuner | Grid | Random | TPE | TimeTPE |
| win | 1 | 1 | **3** | 0 |
| loss | 4 | 4 | **2** | 5 |
| $\overline{Acc_h}$ [%] | 97.378 | 97.866 | **97.967** | 96.881 |
| std | 0.399 | 0.245 | **0.238** | 0.547 |
| $\overline{bias}$ [%] | -0.013 | -0.013 | -0.011 | -0.115 |
| std | 0.0857 | 0.0552 | 0.0812 | 00278 |
| $\bar{t}_{st}$ [s] | 4646 | 4810 | 4595 | 3921 |
| min | 4601 | 4609 | 4289 | 3814 |
| max | 4735 | 4958 | 5084 | 4017 |
| $\bar{t}_{tr}$ | 140 | 134 | 122 | **63.1** |
| best HPs | TPE @98.183% sc=standard, k=rbf, C=13.6, $\gamma$=2.59$\gamma_0$, $\epsilon$=7.62e-8 | | | |

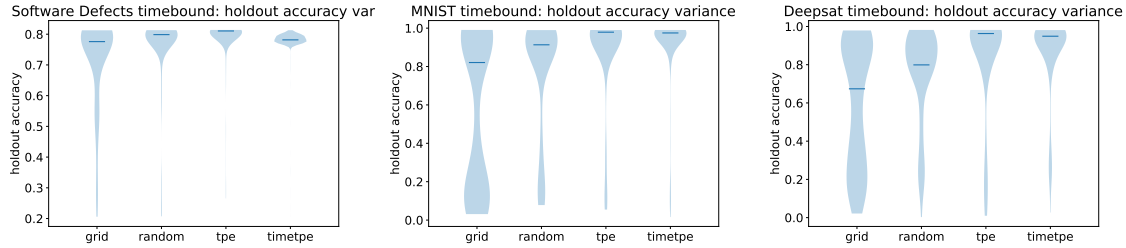
**Table 4.8:** Performance and runtime comparison table B of search space 4 using a timebound stop criterion.

case: The combined metric (e.g. time-metric in this case) is used to guide the Bayesian Surrogate Model. In this way, many different combinations are tested. Then, a purely performance-based metric, such as accuracy, is used to select the model.



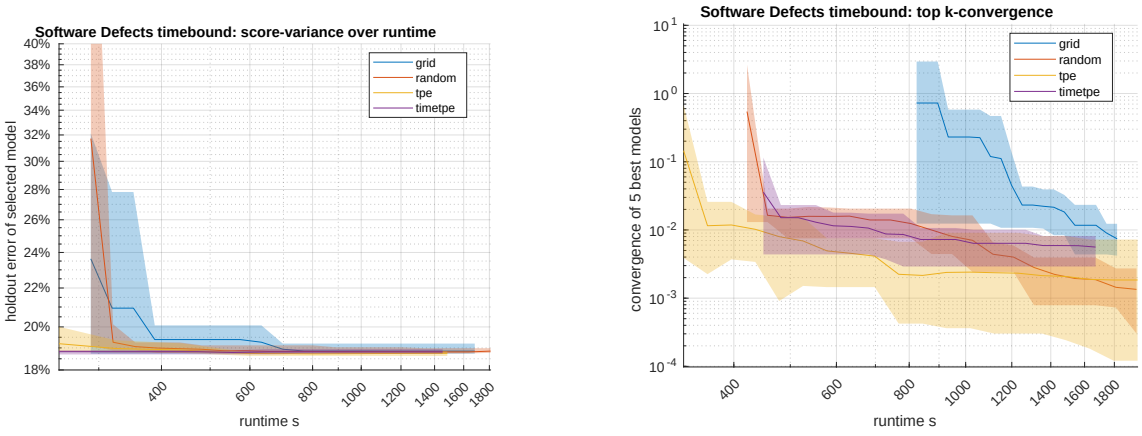

**(a)** **(b)** **(c)**

**Figure 4.20:** Runtime distributions on search space 4 with timebound studies.



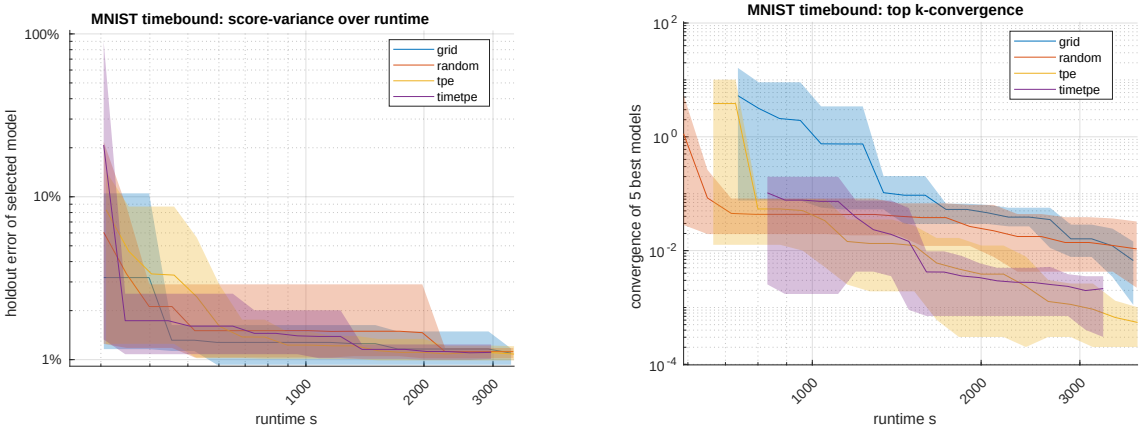

**(a)** **(b)** **(c)**

**Figure 4.21:** Score distributions on search space 4 with timebound studies.



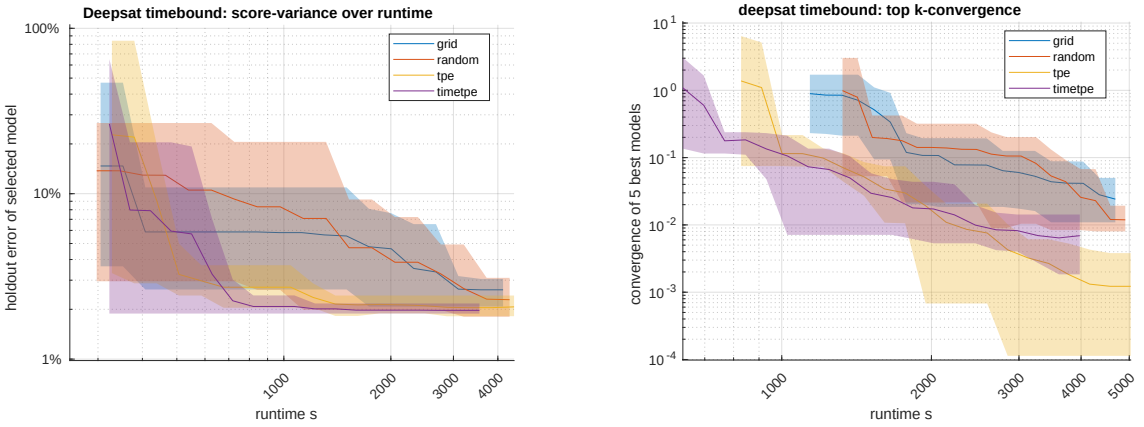

**(a)** **(b)**

**Figure 4.22:** Software Defects score and convergence of search space 4 timebound studies.




**(a)** **(b)**

**Figure 4.23:** MNIST score and convergence of search space 4 timebound studies.




**(a)** **(b)**

**Figure 4.24:** Deepsat Sat6 score and convergence of search space 4 timebound studies.

### 4.4.2 Convergence search

As was the case in the timebound studies, on the convergence search, the accuracy of models found by TimeTPE search is also worse in these tests.

Overall the Grid Search performs poorly in many of the trials, which can be seen in Figure 4.26. It is possible, that this is due to the added search axes. Both coef0 and tol for larger coef0 values and smaller tol values respectively, which can be seen in Figure 3.7. It is likely, that the bad trials were due to a timeout, when fitting the combination. In fact, Grid Search had many more trials with a longer runtime than any other optimizer, which can be seen in Figure 4.25. Despite this, Table 4.9 shows, that Grid Search found well working models. This is most likely due to the way these optimizers behave: A TPE study often converges quickly, as can be seen in Figures 4.27 to 4.29. However, when it does converge, it may only have converged to a local optimum. In this case, it can still explore and find a better optimum. Yet this criterion has stopped the search already. This is likely the case, as the runtime of TPE was always at the lower end. In contrast, the Grid Search takes

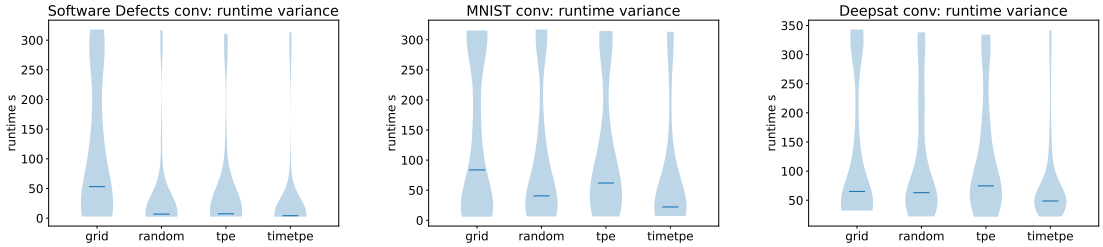
| Dataset | Software Defects | | | | MNIST | | | |
|---|---|---|---|---|---|---|---|---|
| Tuner | Grid | Random | TPE | TimeTPE | Grid | Random | TPE | TimeTPE |
| win | **3** | 1 | 1 | 0 | 0 | **5** | 0 | 0 |
| loss | **2** | 4 | 4 | 5 | 5 | **0** | 5 | 5 |
| $\overline{Acc_h}$ [%] | **81.235** | 81.165 | 81.083 | 79.099 | 98.836 | **98.968** | 98.892 | 96.328 |
| std | **0.0541** | 0.1464 | 0.237 | 1.28 | 0.130 | 0.0277 | **0.0277** | 0.890 |
| $\overline{bias}$ [%] | 0.083 | 0.087 | 0.211 | 0.176 | 0.242 | 0.135 | 0.238 | -0.159 |
| std | 0.0439 | 0.104 | 0.0950 | 0.138 | 0.091 | 0.0520 | 0.0446 | 0.110 |
| $\bar{t}_{st}$ [s] | 1130 | 609 | **550** | 782 | 4807 | 3083 | 1719 | **1387** |
| min | 964 | 117 | 59.3 | 398 | 3507 | 1299 | 1122 | 940 |
| max | 1312 | 1434 | 884.2 | 1244 | 5938 | 4800 | 2452 | 1839 |
| $\bar{t}_{tr}$ | 91.7 | **33.3** | 41.1 | 36.2 | 140 | 103 | 116 | **97.6** |
| best HPs | Random @81.294% sc=standard, k=rbf, C=0.0716, $\gamma$=0.953$\gamma_0$, $\epsilon$=2.10e-7 | | | | Random @98.99% sc=uniform, k=rbf, C=5.45, $\gamma$=9.02$\gamma_0$, $\epsilon$=4.21e-9 | | | |

**Table 4.9:** Performance and runtime comparison table A of search space 4 using a convergence stop criterion.

| Dataset | Deepsat Sat6 | | | |
|---|---|---|---|---|
| Tuner | Grid | Random | TPE | TimeTPE |
| win | 1 | **2** | **2** | 0 |
| loss | 4 | **3** | **3** | 5 |
| $\overline{Acc_h}$ [%] | 97.630 | 97.916 | **97.996** | 96.613 |
| std | 0.338 | 0.381 | **0.288** | 1.24 |
| $\overline{bias}$ [%] | -0.044 | -0.024 | -0.048 | -0.032 |
| std | 0.0410 | 0.00674 | 0.0202 | 0.0978 |
| $\bar{t}_{st}$ [s] | 8183 | 4868 | **2120** | 2488 |
| min | 5607 | 2273 | 1457 | 2049 |
| max | 9474 | 8097 | 3106 | 2865 |
| $\bar{t}_{tr}$ | 132 | 118 | 123 | **82.6** |
| best HPs | Random @98.198% sc=standard, k=rbf, C=11.4, $\gamma$=2.41$\gamma_0$, $\epsilon$=1.81e-9 | | | |

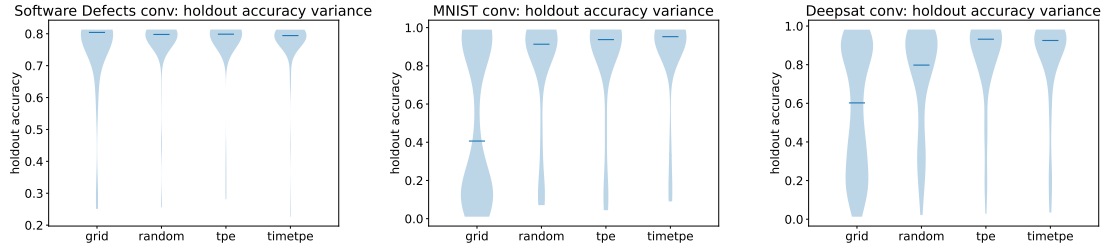
**Table 4.10:** Performance and runtime comparison table B of search space 4 using a convergence stop criterion.

more time to converge and during this time, it is more likely to find a better optimum. To get more comparable results, the values of the convergence search needs to be adjusted, depending on the optimizer. A TPE optimizer can be set to a higher k-value, or a lower threshold, compared to Grid Search, or Random Search. Furthermore, in the case of MNIST, the same reasons are suspected, as in subsection 4.3.3, that TPE had trouble exploiting the optimum closer to the boundary.



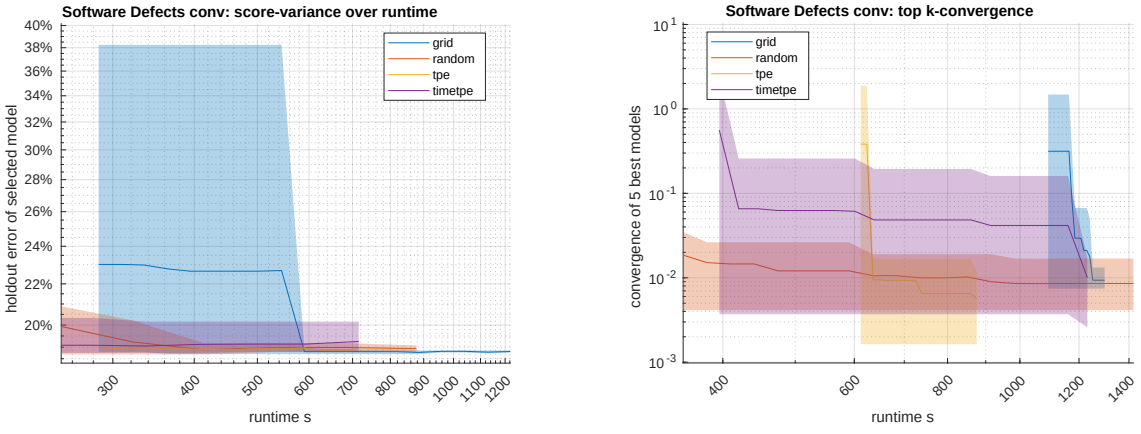

(a) (b) (c)

**Figure 4.25:** Runtime distributions on search space 4 with convergence studies.



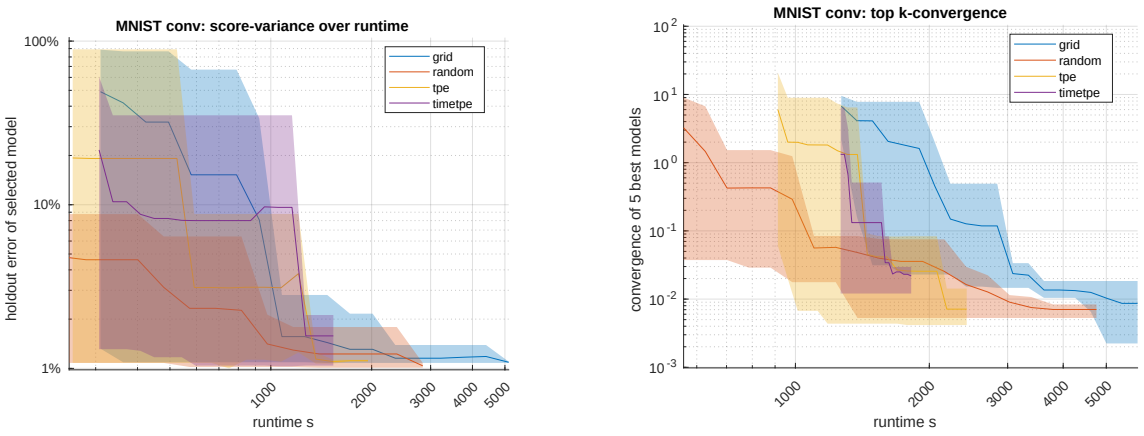

(a) (b) (c)

**Figure 4.26:** Score distributions on search space 4 with convergence studies.



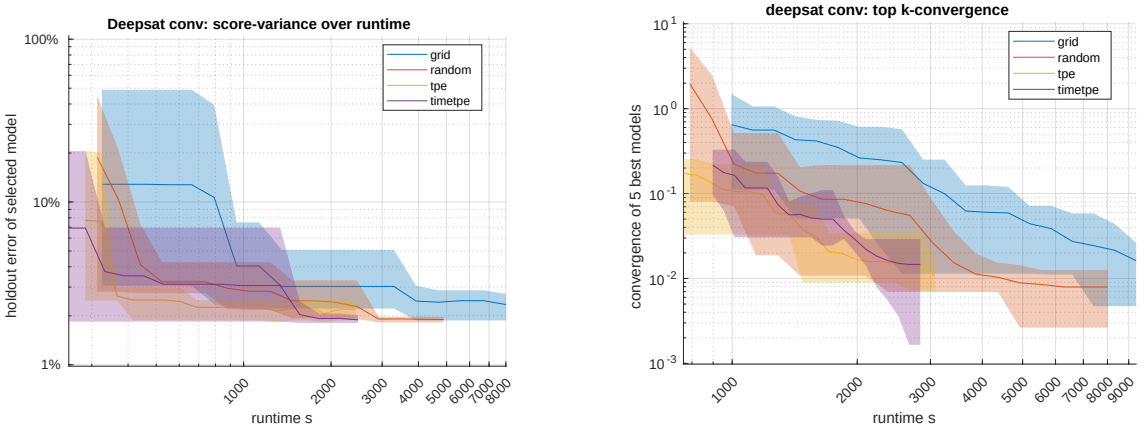

(a) (b)

**Figure 4.27:** Software Defects score and convergence of search space 4 convergence studies.




**(a)** **(b)**

**Figure 4.28:** MNIST score and convergence of search space 4 convergence studies.




**(a)** **(b)**

**Figure 4.29:** Deepsat Sat6 score and convergence of search space 4 convergence studies.

## 4.5 Result Overview

The best results of each search space are compared to the literature in Table 4.11. Using a SVM on the MNIST dataset, our work is able to reproduce the results found by LeCun et al. The average models found from search space 2 on match the performance of the unmodified SVM. However, it cannot match the performance of the Virtual SVM, compared in their work. This is to be expected, since this is a SVM specifically designed for this task, while PLSSVM is a general purpose SVM implementation. The same goes for the Convolutional Neural Network (CNN) presented in [SSP03].

In the case of Deepsat, unfortunately no published performance figures for SVMs was found. The models found in this work were able to exceed the performance of Neural Network (NN) originally developed for Deepsat [BGM+15]. It should come as no surprise, that our SVM models could not match the a custom built CNN described in [LBG+20]. In any case, the performance of the models found is still respectable, given the fact, that dataset like Deepsat are usually not modeled using a general purpose SVMs.

| Dataset | | MNIST | |
|---|---|---|---|
| Search space | Algorithm | $\overline{Acc_h}$ [%] | total time [s] |
| Default values | | 97.95 | 19.8 |
| 0 | Grid Search | 98.33 | 698 |
| 1 | Grid Search | 98.87 | 4216 |
| 2 | Grid Search | 98.93 | 18841 |
| 3 | Grid Search | 98.96 | 48443 |
| 4 timebound | TPE | 98.918 | 3630 |
| 4 convergence | Random Search | 98.968 | 3080 |
| | SVM [LBBH98] | 98.9 | |
| | V-SVM [LBBH98] | 99.2 | |
| | LeNet5 [LBBH98] | 99.05 | |
| | CNN [SSP03] | 99.6 | |
| Dataset | | Deepsat Sat6 | |
| Search Space | Algorithm | $\overline{Acc_{val}}$ [%] | total time [s] |
| Default values | | 95.35 | 49.5 |
| 0 | Grid Search | 97.384 | 766 |
| 1 | TPE | 98.029 | 5634 |
| 2 | Random Search | 98.031 | 15162 |
| 3 | TPE | 98.199 | 44274 |
| 4 timebound | TPE | 97.976 | 4595 |
| 4 convergence | TPE | 97.996 | 2120 |
| | NN [BGM+15] | 93.916 | |
| | CNN [LBG+20] | 99.84 | 1200 |

**Table 4.11:** Performance of found models compared to literature.

# 5 Conclusion and Outlook

In this research the qualitative difference between popular optimizers was compared in different test scenarios. The choice of which optimizer to choose depends on the dataset, the search space, and the objective. If the aim is to find a good solution quickly, then TPE or Random Search seems to be a good choice. The main difference between TPE and Random Search seems to be, that TPE has a lower variance over multiple studies. If the search space is not too large and the objective is to find the best performing model, then the Grid Search may be a viable solution. In any case, the second important choice is, which stop criterion to use. A trial limit is perfect for Grid Search, if on a budget however, then choosing a runtime limit, or a convergence criterion might be a better option. As was demonstrated by the TimeTPE studies, multi-objective optimization can be used with optimizers, such as TPE, to improve a multiple aspects of a model. However, it was found that combining multiple metrics in this way is likely to result in a lower metric than a single-objective optimization with a single metric. More research would be required to determine, how best to use multi-objective optimization for SVMs.

## 5.1 Iterative problem

Running a tuning study is an iterative problem. Finding good search spaces (section 3.4) to test and the correct timeout for the studies (section 3.7) requires retesting the experiments several times. As discussed in subsection 4.3.3, in some of the search axes, the boundary was found as a optimum. In these cases, it might be a good idea to increase the size of the search axes. It was also observed that the optimizer never selected linear models. With this result in mind, it would be a good idea, to remove this combination from the search space, to maximize the the optimizer's efficiency. Since it is difficult to predict such behavior, one strategy is to run a search, evaluate the results, and then modify the search space accordingly. Tuning the search space was not the aim of this work, but it would be an interesting future research direction.

## Outlook

## 5.2 Bias

During this research it became apparent (subsection 4.3.3) that the bias often increases the difficulty of any optimizer to find a well generalizing model. Therefore, future work should focus on reducing the bias of any optimizer. For applications like ours that deal with large datasets, it would be desirable to implement a method that reduces the bias but has a smaller resource impact than k-fold cross-validation.

## 5.3 Other configurations

To get a more general idea of how the optimizers behave with PLSSVM, it would be a good idea to increase the number of datasets being compared. Also, it might be a good idea to run these experiments on different hardware, for example different GPUs. In this context, a very important test to perform is to analyze the energy used by the CPU. Also, tuning PLSSVM with different software backends and OSs would be required, to strengthen our findings.

# Bibliography


[AF91] S. Aeberhard, M. Forina. *Wine*. UCI Machine Learning Repository. 1991. DOI: 10.24432/C5PC7J (cit. on p. 29).

[AMS+21] M. M. Ahsan, M. A. P. Mahmud, P. K. Saha, K. D. Gupta, Z. Siddique. "Effect of Data Scaling Methods on Machine Learning Algorithms and Model Performance". In: *Technologies* 9.3 (2021). ISSN: 2227-7080. DOI: 10.3390/technologies9030052 (cit. on pp. 31, 33, 48).

[ASY+19] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama. *Optuna: A Next-generation Hyperparameter Optimization Framework*. 2019. arXiv: 1907.10902 [cs.LG] (cit. on pp. 3, 13, 19, 36).

[BB12] J. Bergstra, Y. Bengio. "Random search for hyper-parameter optimization". In: *J. Mach. Learn. Res.* 13.0 (2012), 281–305. ISSN: 1532-4435 (cit. on p. 22).

[BBL+21] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, D. Deng, M. Lindauer. *Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges*. 2021. arXiv: 2107.05847 [stat.ML] (cit. on pp. 17–19, 24).

[BGM+15] S. Basu, S. Ganguly, S. Mukhopadhyay, R. DiBiano, M. Karki, R. Nemani. *DeepSat - A Learning framework for Satellite Imagery*. 2015. arXiv: 1509.03602 [cs.CV] (cit. on pp. 34, 35, 78).

[BS02] H.-G. Beyer, H.-P. Schwefel. "Evolution strategies - A comprehensive introduction". In: *Natural Computing* 1 (Mar. 2002), pp. 3–52. DOI: 10.1023/A:1015059928466 (cit. on p. 24).

[CL11] C.-C. Chang, C.-J. Lin. "LIBSVM: A library for support vector machines". In: *ACM Trans. Intell. Syst. Technol.* 2 (2011), 27:1–27:27. DOI: 10.1145/1961189.1961199 (cit. on p. 26).

[CSB15] W. Czarnecki, S. Smusz, A. Bojarski. "Robust optimization of SVM hyperparameters in the classification of bioactive compounds". In: *Journal of cheminformatics* 7 (Dec. 2015), p. 38. DOI: 10.1186/s13321-015-0088-0 (cit. on p. 30).

[CV95] C. Cortes, V. Vapnik. "Support-Vector Networks". In: *Mach. Learn.* 20.3 (1995), 273–297. ISSN: 0885-6125. DOI: 10.1023/A:1022627411411 (cit. on p. 25).

[Dzu24] Y. Dzubba. *PLSSVM-HPO*. 2024. URL: https://gitlab-sim.informatik.uni-stuttgart.de/domanspr/plssvm_hpo (cit. on p. 13).

[FH19] M. Feurer, F. Hutter. "Hyperparameter Optimization". In: *Automated Machine Learning*. 2019, pp. 3–33. DOI: 10.1007/978-3-030-05318-5_1 (cit. on p. 24).

[Fis36] R. A. Fisher. "THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS". In: *Annals of Eugenics* 7.2 (1936), pp. 179–188. DOI: 10.1111/j.1469-1809.1936.tb02137.x (cit. on p. 29).

[GBV20] M. Grandini, E. Bagli, G. Visani. *Metrics for Multi-Class Classification: an Overview*. 2020. arXiv: 2008.05756 [stat.ML] (cit. on pp. 15, 16, 41).

[GMHL20] E. C. Garrido-Merchán, D. Hernández-Lobato. "Dealing with categorical and integer-valued variables in Bayesian Optimization with Gaussian processes". In: *Neurocomputing* 380 (Mar. 2020), 20–35. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2019.11.004 (cit. on p. 23).

[HMW+20] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2 (cit. on pp. 40, 43).

[HS52] M. R. Hestenes, E. Stiefel. "Methods of conjugate gradients for solving linear systems". In: *Journal of research of the National Bureau of Standards* 49 (1952), pp. 409–435. DOI: 10.6028/jres.049.044 (cit. on p. 26).

[JSW98] D. R. Jones, M. Schonlau, W. J. Welch. "Efficient Global Optimization of Expensive Black-Box Functions". In: 13.4 (1998), 455–492. ISSN: 0925-5001. DOI: 10.1023/A:1008306431147 (cit. on p. 23).

[KFB+17] A. Klein, S. Falkner, S. Bartels, P. Hennig, F. Hutter. *Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets*. 2017. arXiv: 1605.07079 [cs.LG] (cit. on p. 30).

[KK20] M. Kuhn, J. Kjell. *Feature Engineering and Selection: A Practical Approach for Predictive Models*. Chapman & Hall, 2020. URL: http://www.feat.engineering/ (cit. on pp. 31–33).

[LBBH98] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. Vol. 86. 11. IEEE, 1998, pp. 2278 –2324. DOI: 10.1109/5.726791 (cit. on pp. 30, 33, 34, 78).

[LBG+20] Q. Liu, S. Basu, S. Ganguly, S. Mukhopadhyay, R. DiBiano, M. Karki, R. Nemani. "DeepSat V2: feature augmented convolutional neural nets for satellite image classification". In: *Remote Sensing Letters* 11.2 (2020), pp. 156–165. DOI: 10.1080/2150704X.2019.1693071 (cit. on pp. 34, 78).

[LLN+18] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gozalez, I. Stoica. "Tune: A Research Platform for Distributed Model Selection and Training". In: *CoRR* abs/1807.05118 (2018). arXiv: 1807.05118 (cit. on p. 19).

[MG13] N. Mehra, S. Gupta. "Survey on multiclass classification methods". In: *Int. J. Comput. Sci. Inf. Technol.* 4 (Jan. 2013), pp. 572–576 (cit. on p. 28).

[MRV+15] R. Mantovani, A. Rossi, J. Vanschoren, B. Bischl, A. de Carvalho. "Effectiveness of Random Search in SVM hyper-parameter tuning". In: July 2015. DOI: 10.1109/IJCNN.2015.7280664 (cit. on p. 29).

[PVG+12] 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, E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *CoRR* abs/1201.0490 (2012). arXiv: 1201.0490 (cit. on pp. 19, 28).

[Pla98] J. Platt. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Tech. rep. MSR-TR-98-14. Microsoft, 1998. URL: https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/ (cit. on p. 26).

[RDPCV+18] A. Rojas-Domínguez, L. C. Padierna, J. M. Carpio Valadez, H. J. Puga-Soberanes, H. J. Fraire. "Optimal Hyper-Parameter Tuning of SVM Classifiers With Application to Medical Diagnosis". In: *IEEE Access* 6 (2018), pp. 7164–7176. DOI: 10.1109/ACCESS.2017.2779794 (cit. on pp. 25, 29, 70).

[Ros95] G. van Rossum. *Python tutorial*. Tech. rep. CS-R9526. Amsterdam: Centrum voor Wiskunde en Informatica (CWI), 1995 (cit. on p. 36).

[SSP03] P. Y. Simard, D. Steinkraus, J. Platt. "Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis". In: IEEE, 2003. DOI: 10.1109/ICDAR.2003.1227801 (cit. on pp. 34, 78).

[SSWB00] B. Scholkopf, A. Smola, R. C. Williamson, P. L. Bartlett. "New Support Vector Algorithms". In: *Neural Computation* 12 (2000), pp. 1207–1245. DOI: 10.1162/089976600300015565 (cit. on p. 29).

[SV99] J. Suykens, J. Vandewalle. "Least Squares Support Vector Machine Classifiers". In: *Neural Processing Letters* 9 (June 1999), pp. 293–300. DOI: 10.1023/A:1018628609742 (cit. on p. 26).

[Shl14] J. Shlens. *A Tutorial on Principal Component Analysis*. 2014. arXiv: 1404.1100 [cs.LG] (cit. on p. 34).

[Tha19] A. Tharwat. "Parameter investigation of support vector machine classifier with kernel functions". In: *Knowledge and Information Systems* 61.3 (Dec. 2019), pp. 1269–1302. DOI: 10.1007/s10115-019-01335-4 (cit. on p. 17).

[US 09] US department of agriculture. *National Agriculture Imagery Program*. 2009. URL: https://www.fsa.usda.gov/Internet/FSA_File/naip_2009_info_final.pdf (cit. on p. 35).

[VCBP22] A. Van Craen, M. Breyer, D. Pflüger. "PLSSVM: A (multi-)GPGPU-accelerated Least Squares Support Vector Machine". In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022, pp. 818–827. DOI: 10.1109/IPDPSW55747.2022.00138. URL: https://github.com/SC-SGS/PLSSVM (cit. on pp. 3, 13, 26, 36, 48).

[WGL16] A. Wan, D. Gosh, S. Liu. *A Guide to MNIST*. 2016. URL: https://fsix.github.io/mnist/ (cit. on p. 34).

[WKT01] T. Wakahara, Y. Kimura, A. Tomono. "Affine-invariant recognition of gray-scale characters using global affine transformation correlation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23.4 (2001), pp. 384–395. DOI: 10.1109/34.917573 (cit. on p. 35).

[WR23] A. C. Walter Reade. *Binary Classification with a Software Defects Dataset*. 2023. URL: https://kaggle.com/competitions/playground-series-s3e23 (cit. on p. 33).

[Wat23] S. Watanabe. *Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance*. 2023. arXiv: 2304.11127 [cs.LG] (cit. on p. 23).

[YAA24] Z. Yang, K. Adamek, W. Armour. *Part-time Power Measurements: nvidia-smi's Lack of Attention*. 2024. arXiv: 2312.02741 [cs.DC] (cit. on pp. 55, 56).

[YB19] C. Yadav, L. Bottou. *Cold Case: The Lost MNIST Digits*. 2019. arXiv: 1905.10498 [cs.LG] (cit. on p. 35).

[YJG03] A. B. Yoo, M. A. Jette, M. Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. Feitelson, L. Rudolph, U. Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4. DOI: 10.1007/10968987_3 (cit. on p. 53).

All links were last followed on April 11, 2024.

# Appendix

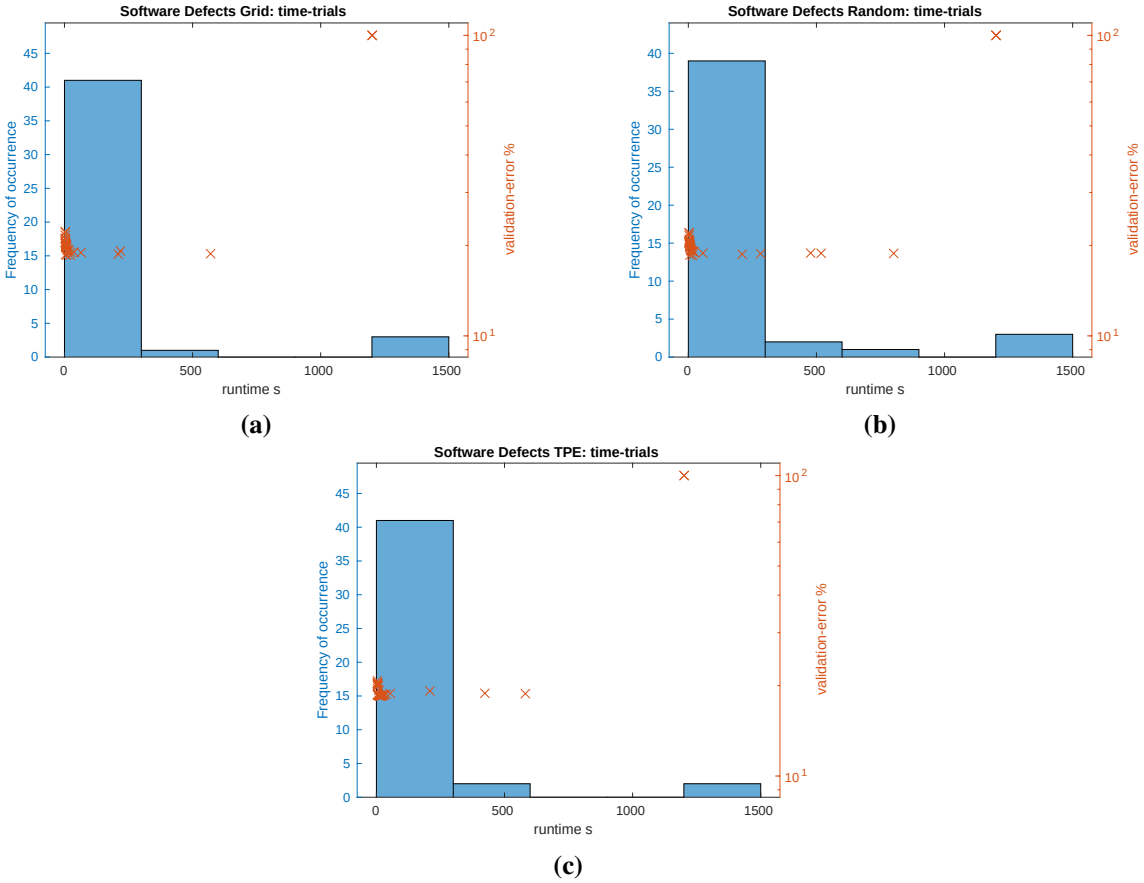



**Figure 1:** Time trials of the Software Defects dataset (section 3.2). Time trials of MNIST can be found in Figure 3.8.

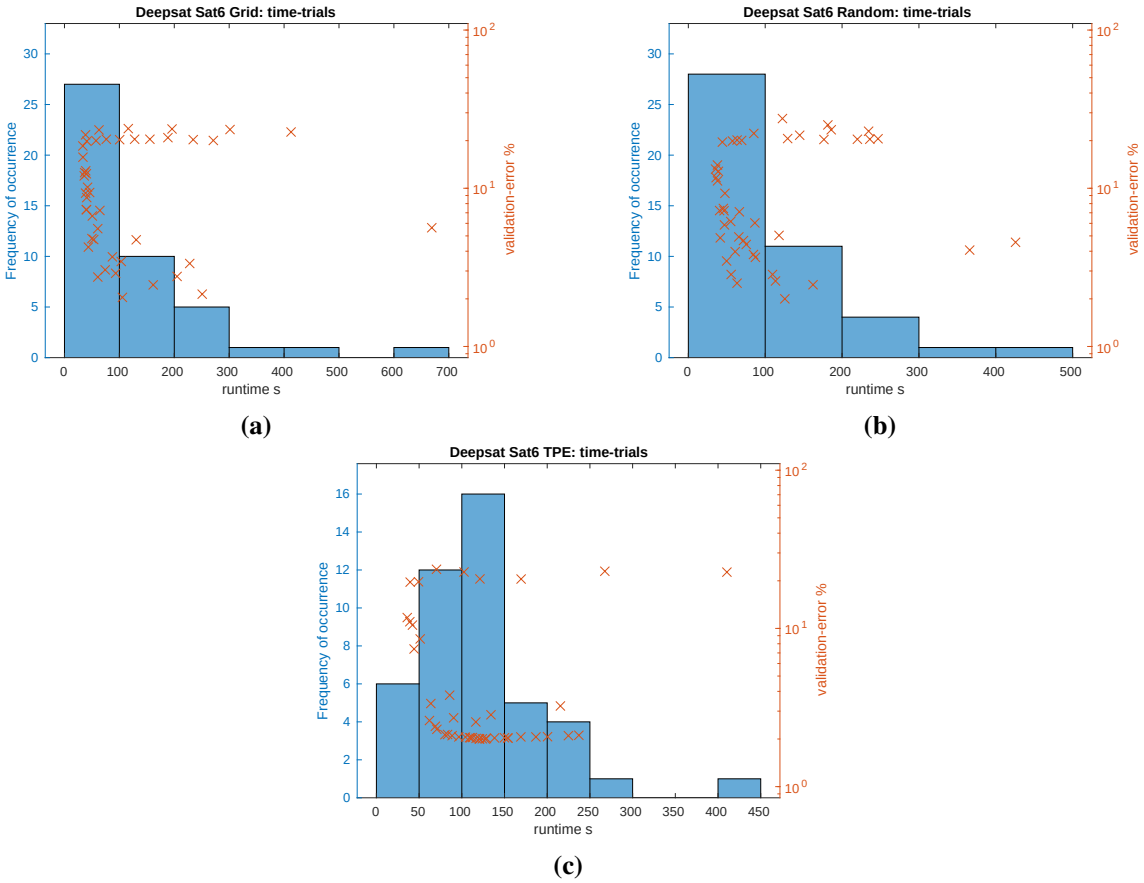
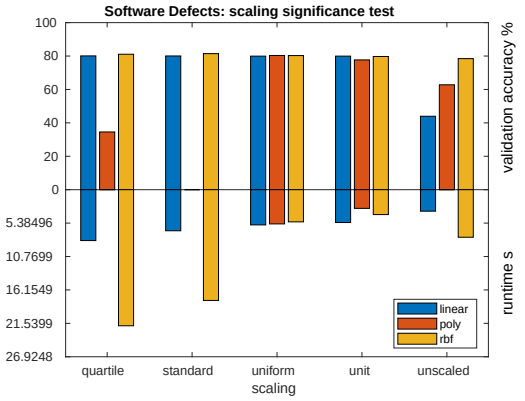
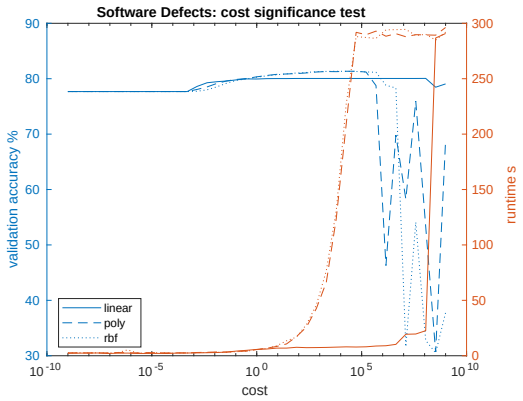


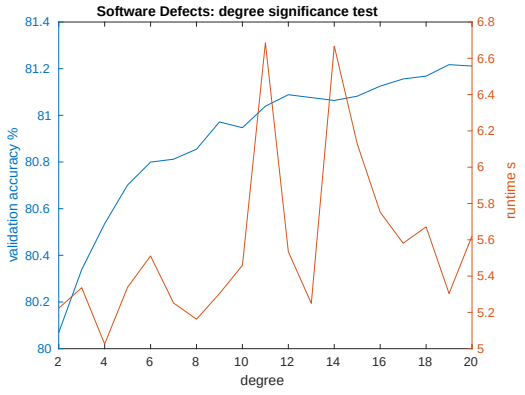

**Figure 2:** Time trials of the Software Defects dataset (section 3.2). Time trials of MNIST can be found in Figure 3.8.



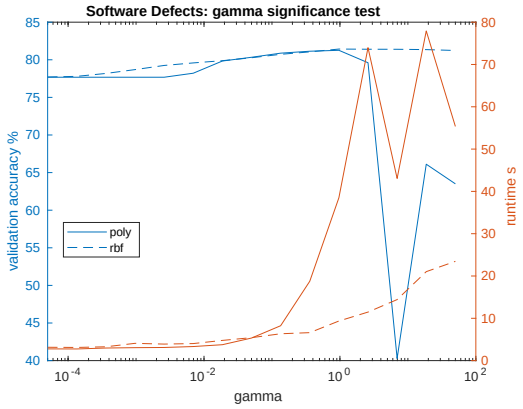

**(a)** Fitting polynomial kernel with standard scaled data timed out and quartile scaled data is outside the scale at 120 s.



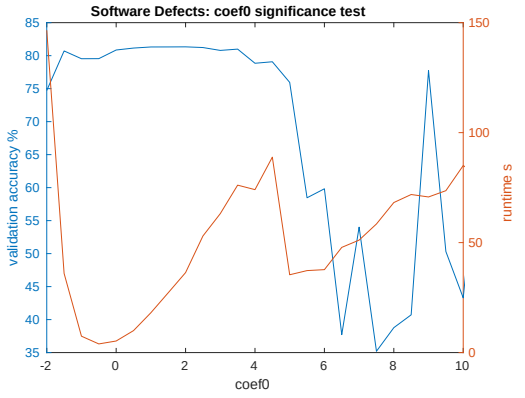

**(b)**



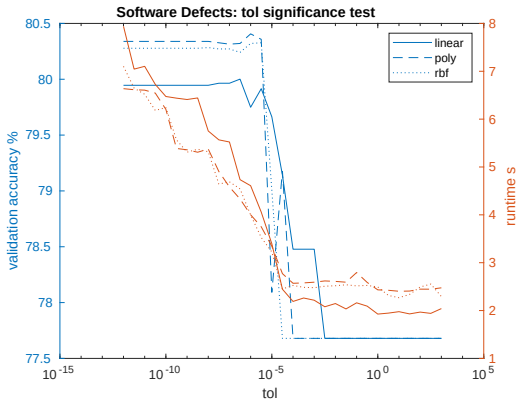

**(c)**




**(d)**




**(e)**




**(f)**

**Figure 3:** Significance tests of Software Defects dataset (section 3.2). Test score refers to the Accuracy on the validation slice of the dataset, as defined in section 2.2. MNIST significance test can be found in Figure 3.7.

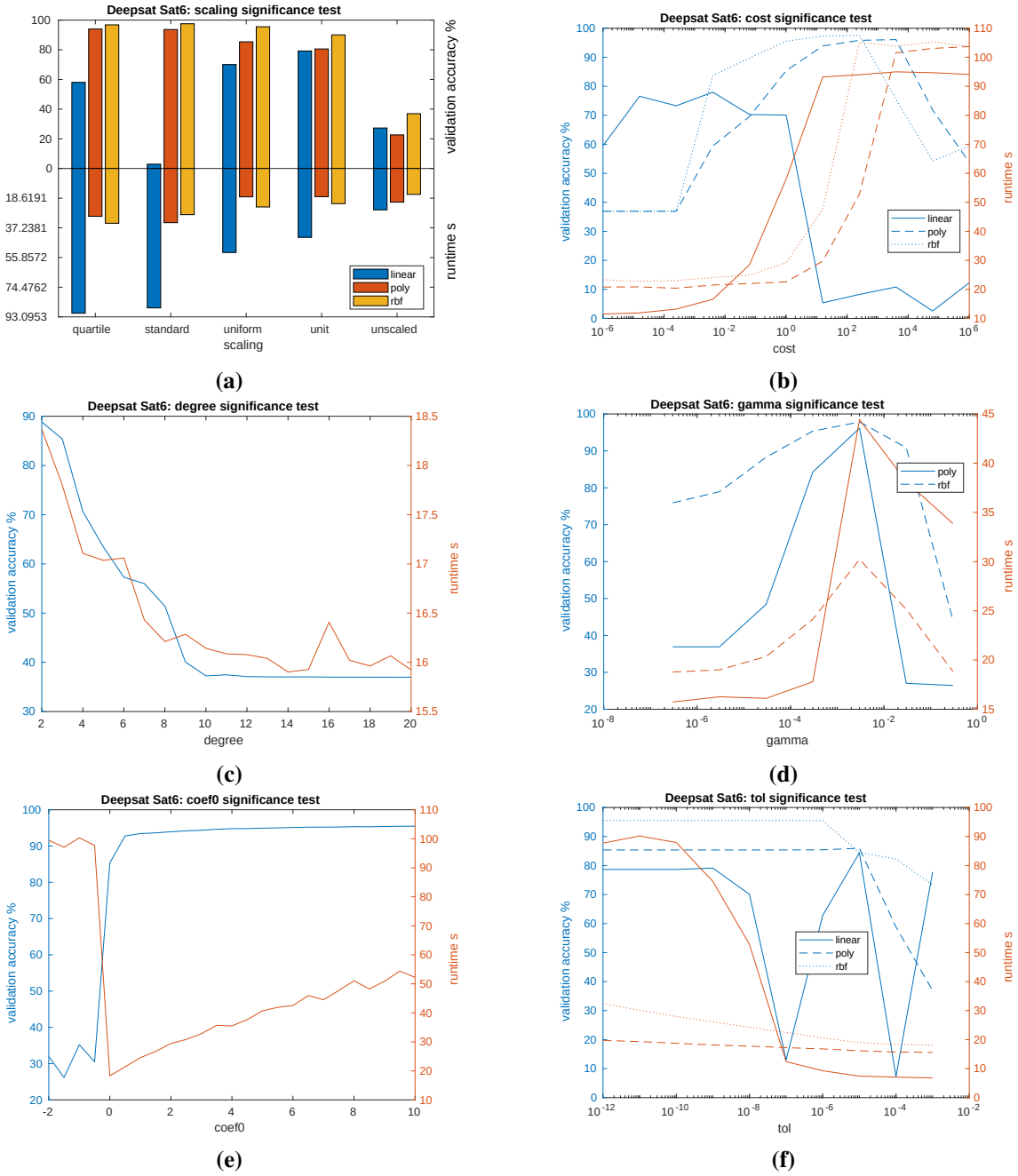



**Figure 4:** Significance tests of Deepsat Sat6 dataset (section 3.2). Test score refers to the Accuracy on the validation slice of the dataset, as defined in section 2.2. MNIST significance test can be found in Figure 3.7.

**Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature