University of Bamberg
Faculty of Information Systems and Applied Computer Science
Chair of Mobile Systems

Master's Degree in Computing in the Humanities

Master's Thesis

Balancing Performance and Resource-Awareness:

Optimizing the Model Selection and Ensemble Process in Machine Learning

Author

Niklas Diller

Reviewers

Prof. Dr. Daniela Nicklas
Michael Freitag

Bamberg, July 11, 2024

Zusammenfassung

TODO

Content

1 Introduction

A common and widely known challenge in training and using machine learning models is to find a suitable balance between overfitting and underfitting [1]. The goal for coming up with valuable predictors for a specific problem therefore is often to create models that neither simply reproduce results gathered from the training data, nor make too broad generalizations. Overfitting would mean that the model would not only learn the underlying pattern of data, but also random noise that comes with it. This results in a bad performing model for new and unseen data, as the model is constantly trying to match its' predictions with the training data too closely. Underfitting on the other hand happens if a model fails to completely cover the complexity of a specific problem. Because new data points are unfamiliar to the model, the resulting predictions will also be poor and not accurate [2]. It becomes a crucial task for machine learning engineers to balance the amount of the training data and the hyperparameters for machine learning models to find a suitable golden mean of creating models that correctly represent the underlying patterns but is at the same time capable of handling new data points.

A similar crucial balancing act in machine learning that seems to find much less attention, however, is the equipoise of the concepts of performance and resource awareness. Both principles are essential for a valuable machine learning model: Predictors need to produce accurate results in order to be of value and research has laid and increasingly focus on designing resource aware machine learning systems [3]. While the two concepts don't necessarily cancel each other out, there seems to be a trade-off between them when exploring and training models. Good performing models often make use of a big set of features and complex algorithms and in order to produce a stream of up-to-date predictions, they are often called in short time intervals. Machine learning models of this kind, however, tend to demand a larger number in computational resources, e.g. CPU time, memory consumption or network utilization. When only looking for the best performing model, i.e. the model that produces the most accurate results, the predictions might become too expensive in regards to memory or battery usage, or simply too computationally complex [4]. Especially in the context of edge computing, this balance between good performance and good resource utilization becomes crucial. Edge computing is a computing paradigm that focuses on processing data at the same place where it is produced: at the edge of the network. In contrast to cloud computing, which works by sending all raw data to a central node to be processed there, edge computing operates on the data near the sensors that produce it. That way, less data needs to be sent on a anyway limited network bandwidth, accounting for faster response times, and less network pressure [5].

The main motivation for this work are two projects that the Chair of Mobile Systems of the University of Bamberg is carrying out. Both raise the demand of a model set retrieval system that lets the user decide, how significant resource awareness is supposed to be in relation to performance. These projects will be introduced in greater detail in the following chapter. For a model retrieval system to be working, several varying machine learning models need to exist first. It was therefore decided to implement a model training pipeline along with the retrieval system itself. As many questions regarding the conceptualization of different parts of the use cases are still open, this work also aims to elaborate various theoretical ideas and concepts regarding resource awareness, prediction horizons and model scores.

This work will focus on answering the question on how to manage a meaningful balance between performance and resource awareness in the model selection process. In particular the creation of

ensembles or model sets will be discussed from this perspective while applying them to the use cases that this work will refer to. The chapters of this thesis are structured as followed:

2 Related Work

This chapter aims to give an overview of the theoretical background of the topics and concepts that will be examined and discussed in this work. Before going deeper into the topic of the model selection process and how to optimize it, certain keywords need to be specified in order to have a common understanding about reoccurring terms. For this, the definitions of Zhi-Hua Zhou are used [6]. The process of applying a learning algorithm to data can therefore be considered as training. By doing that, a model or a so-called learner is generated, that is then used to predict the outcome of a specific problem. A distinction can be made between supervised and unsupervised learning. The former uses predefined labels or values as possible outcomes and is used in the use cases of both parking availability prediction as well as cattle activity recognition. In this case, the model can also be called predictor. As the labels in the cattle activity recognition use case are of categorical type, another name that can be used for the model is classifier. Unsupervised learning on the other hand is working without the use of any fixed labels or categories. Its aim is to uncover specific traits and structure in the data.

Model selection itself now describes the process of selecting the best learning algorithm – for example Linear Regression or Decision Tree – and tuning its corresponding parameters like Data Preprocessing or maximum depth [6]. To improve both the performance and the usability of the model management framework, this thesis will focus on the use of ensemble methods. This chapter will give an introduction to several theoretical concepts and paradigms. First, ensembles in machine learning will be introduced. Next, this chapter will give an overview of the use cases that this work refers to. Subsequently, both resource awareness as well as performance will be presented.

2.1 Ensembles

Given that the output of a learning algorithm that works on a certain training set is considered a classifier, an ensemble can generally be defined as a set of classifiers. Each individual output of these classifiers is then combined in order to achieve one decision that predicts the problem in a better way, than the single classifiers could have done on their own and in an isolated way [7]. Ensembles can generally be divided into homogenous and heterogenous ones. While the former consist of models that have been trained on the same kind of machine learning algorithm, the latter is constructed using different kind of algorithms in the ensemble, for example both decision trees and SVMs [6]. Three main reasons can be identified on why ensembles are often working better in producing high quality predictions in comparison to individual classifiers [7]: First off, an ensemble gives a statistical advantage in reaching the true hypothesis of the observed problem. By averaging the classifiers in use, the distance from the true hypothesis to the hypothesis used by the model can be reduced. Secondly, computing the unknown true hypothesis is often difficult. That is why using distinct approaches coming from different classifiers could help to come closer to the true function. Lastly, representation plays a role, as it is often not able to represent the true hypothesis by using the available hypothesis space. By combining different classifiers, this space of representable functions can be widened. Rokach identifies four building blocks that make up an ensemble [8]: A training set that the model will learn on, a base inducer that forms a classifier by obtaining the training set, a diversity generator that provides the required variety in the classifiers and lastly a combiner that merges the classifications of the models into one single prediction. In this work, the

Following, a more detailed look into individual ensemble methods will be made. By doing that, the ensemble methods will be divided into dependent and independent methods.

2.2 Dependent and Independent Methods

3 Design and Architecture

For a model set retrieval system to be working properly, of course, there need to be machine learning models to choose from in the first place. Therefore, next to the design and implementation of the actual Top-k algorithms, a diverse and extensive model database first had to be established. This chapter on architecture and design, as well as the following chapter that focuses on implementation, will describe the development process of both a Top-k retrieval system as well as a training pipeline that was established to automatically preprocess data and train machine learning models based on this data.

3.1 Training Pipeline

The first section that was worked on when starting this project, is the training pipeline. First versions of a model trainer have already been developed by fellow students Paul Pongratz and Alexandr Litvin in previous semesters. This original form of a module that could train machine learning models for the use cases introduced in chapter 2 served as a valuable basis to improve upon. The model trainer was written in Java and used the Weka (Waikato Environment for Knowledge Analysis) [9] library as its main building block. Weka is an open-source software that specializes in machine learning and data mining. With the help of this library, data entries of the parking behavior including time stamps and weather information can be converted into so-called `Instances` which are then used to train different kinds of classifiers such as Linear Regression or Random Forest which are also managed by Weka. In addition, the library was used to apply filters, in order to make use of different feature-scaling methods. To get the relevant parking data that is stored in a `postgres` database on the university's servers, a Java `DriverManager` of the library `java.sql` is used. That way a connection to the `postgres` database is established and a SQL statement with the relevant information is both created and executed. Another library that was essential in the creation of the model trainer is `tablesaw`, which makes it possible to create and manage data into `dataframes`, similar to the `pandas` library in Python. `Tablesaw` is mainly used to establish training- and testing-datasets out of the data queried from the database. Some exemplary usages include counting rows, dropping columns, or joining tables.

Regarding the overall architecture, the training pipeline initially was constructed in a way that let the user put in the desired model settings in a `config.properties` file which then was read by an `InputStream` to populate an object of the class Settings. This of course had to be modified in a way, that had the properties file be filled in automatically. Additionally, in the original version of the model trainer, the relevant data was pre-processed for each model, which turned out to be a very time-consuming effort. It was therefore decided to introduce a second entry point to the model trainer java program that takes care of all the pre-processing of the data entries. The pre-processed data is then stored in a separate database, which is later accessed by the actual model trainer for training and testing. The implementation chapter will go into more detail regarding the actual integration of those classes.

3.2 Retrieval System

In contrast to the training pipeline, the Top-k retrieval system was built entirely from scratch. It was integrated into the gradle project as a separate running module and written in Python to make use of certain frameworks and libraries. Mainly because of its lightweight structure and straightforward setup, Flask was chosen as the model set retrieval systems web framework. Flask offered many free choices on what libraries to use in the development stage, without forcing any specific dependencies and therefore bloating the project. The basic principle of the model set retrieval system was to make it easily extendable by also making sure maintenance can be made without the maintainer having to study the entire project first. Using a Representational State Transfer (REST) API, several services have been developed that allow the user to communicate with the retrieval system through their corresponding API endpoint using HTTP. The services of the API provide CRUD (Create, Read, Update, Delete) operations, depending on what request the user has. In the development stage, the API calls have been made using Insomnia, an open-source software that focuses on designing and testing APIs. For testing, first, a virtual environment is activated that takes care of the selection of a Python interpreter and important dependencies. Subsequently, the university network is joined, typically using a VPN. Finally, the Flask app is run on a specific port, that is then reached using the corresponding URL in Insomnia. In production, of course, alternative API testing services can be used, such as SwaggerUI or Postman.

At this time, the following endpoints have been established:

- **Model selection** (`/api/select`): POST operation that retrieves information about a model that the user specifies via a JSON file.
- **Model direct selection** (`/api/model/<model-name>`): READ operation, that retrieves information about a model the user specifies directly in the URL.
- **Top-k single model** (`/api/topk`): CREATE operation that retrieves the k best models based on the users' input.
- **Top-k model set** (`/api/topk/modelsets`): CREATE operation that retrieves the k best model sets based on the users' input.

Probably the most central library that was utilized in the Top-k retrieval system is `pandas`, an open-source project that rose to one of the most contributed-to libraries in Python [10]. Similar to `tablesaw`, `pandas` focuses on managing, manipulating and analyzing data sets. Especially the object `DataFrame` proved to be a very powerful data structure that helped the implementation of the previously presented Top-k algorithms tremendously. To establish a connection to the model database, the `PostgreSQL` database adapter `Psycopg2` was used. Once the connection to the database is successfully made, a previously prepared SQL statement is then executed via a `Psycopg2` object called `cursor`. To save time and computational resources, the model set retrieval system was designed in a way, that only one SQL statement is executed per API call.

The use case of selecting not only single items (i.e. single models) through a top-k retrieval system but rather composing whole sets of those items made it necessary to approach the design of this module with numerous extra steps in mind. Each model set should contain models of different prediction horizons, a metric that will be introduced in detail in the implementation chapter. It is assumed that

each model set contains either two or three different prediction horizons, and therefore two or three different models. The central challenge was to find a way to determine the resource awareness of a model set before actually composing that model set, which ultimately led to the introduction of intra-model resource awareness. In a simple top-k retrieval system, items containing different metrics are sorted for each of those metrics. The items that should be returned in this use case are the model sets, whilst the relevant metrics are the performance (i.e. accuracy or an error metric like MAE) and resource awareness. This resource awareness however has been previously defined to be measured by ascertaining the underlying QSLs in that model set. As mentioned in chapter 2, a straightforward approach would then have been to simply create model sets before retrieving them using a top-k algorithm. The accuracy metric could then be asserted by taking the average value of the models' accuracy values that are contained in the model set, for instance. The resource awareness metric could be easily measured by looking at the shared features of the models in the model set. The problem with this straightforward approach is, that a vast number of model sets would have to be created for it to work properly. Taking into account all the different metrics that a single model can have (e.g. window size, classifier, combination of features, ...), all possible combinations of two or three of those models is a huge number to create. The database containing all possible model sets would have to be iterated through by the top-k algorithms, making the retrieval process slower than using a smaller database. Additionally, new model sets would have to be created every time new models are added to the model database, making it complex to manage and keep the model set database updated.

Because of these reasons stated above, it was decided against using such a predefined model set database. A number of alternative approaches were designed at this point of the project, all approaching the resource awareness issues in different ways. They will shortly be introduced in the following section.

Alternative 1

This approach is defined by neglecting the idea of having QSLs altogether. Instead, the resource awareness of a model set is supposed to be determined by establishing metrics of single models that represent the resource awareness. At this stage in development, the idea of having inter-model resource awareness assessed first came up. The metrics that were identified for this purpose were the number of features or the window size of a model. Using a top-k algorithm, the best models regarding both performance and resource awareness would then be searched and put into model sets. Depending on which prediction horizon is still missing in a model set, the best-rated model of both performance and resource awareness is then added to the set.

Alternative 2

Another possibility would be to calculate the QSLs at the end of the model set creation process. This approach starts the same way as Alternative 1, by finding the best models regarding performance and resource awareness using suitable metrics for either dimension as discussed before (e.g. accuracy and number of features). The single model retrieval would then continue until a model of each prediction horizon is found. (Or k models have been retrieved, with each prediction horizon being selected at least once.) Subsequently, these retrieved models would be combined into every model set possible but under the condition that each model set includes the necessary prediction horizons. Ultimately the QSLs would be calculated and the model sets would be sorted according to their levels.

Alternative 3

This approach is a modification of Alternative 2. It starts by creating lists of all models according to their prediction horizon. For instance, if model sets with three different prediction horizons should be created, three different lists would first be created. Thereafter, the k best models for each prediction horizon would be retrieved and combined into all possible combinations. Ultimately the QSLs would be calculated and the model sets would be sorted according to their levels, just like in Alternative 2. The major difference between Alternative 3 and 2 is the number of models that are considered for model set creation. While in Alternative 2 only a few or maybe only one model could be considered for a certain prediction horizon, the number of models considered per prediction horizon is always the same in Alternative 3.

Alternative 4

This is the option that was ultimately decided to be used in the final implementation of the model set retrieval system. The main idea behind Alternative 4 is to combine the advantages of each of the previously introduced alternatives in order to create a retrieval design that works best for the required use case. For this, intra-model resource awareness metrics and QSLs are both used. First, the model database is split up into lists of each prediction horizon. Then, the k best models for each prediction horizon are retrieved based on a constructed Model Score which is a combination of the models' performance and its intra-model resource awareness metric, using a top-k algorithm. Subsequently, the retrieved models are put into model sets by using every possible combination (whilst also respecting the condition to use a specific prediction horizon only once inside a model set). Another score called Aggregated Model Score is then created, which depicts the summarized Model Scores inside a model set. Afterward, the QSLs for each created model set are calculated. As a final step, a second top-k retrieval is started, in order to acquire the k best model sets based on an overall Model Set Score that is created by weighting both the previously calculated Aggregated Model Score and the QSLs which act as an inter-model resource awareness metric. For this second top-k retrieval, a separate k and weight can be chosen in comparison to the first top-k algorithm, to fine-tune the request if necessary. The k best model sets are then returned to the user, together with all their associated meta information and scores. In summary, the implemented model set retrieval system makes use of running two different types of objects through the top-k algorithms: First, the single models that are later to fill the sets with, and the complete model sets second. The user however does not have to undergo any additional steps in order to obtain the top k model sets besides providing the desired settings – the model set retrieval system manages calling the different algorithm rounds on its own.

The design of the QSLs has been generally adapted from Sünkel et. al. [11]. However, Level 1 and Level 2 have been disregarded, as they describe aspects of alignments that are not applicable for the context of parking prediction: Level 1 of the original QSLs is reached when two models use the same sensor system, while Level 2 is reached if two models have the same preprocessing. As sensor systems are of no relevance in the parking availability prediction use case, Level 1 can never be truly reached and will therefore be ignored in the context of this architecture and implementation. Furthermore, while general preprocessing is in fact happening in the developed system for this work, it is vastly different from the preprocessing in the cattle activity recognition use case: There is no sensor data that has to be combined in a way to get a certain metric, like with the accelerometer signal. This level can therefore also be disregarded. Consequently, the possible QSLs in this work are 0, 3, and 4.

By designing the model set retrieval system in a way, that makes it possible to choose the aggregation function for determining the overall QSL of a model set, the user is given even more freedom in customizing the query. Note, that this is only relevant for model sets containing at least three models, as a set containing only 2 models does only have one QSL to be determined. Model sets with three models, however, make it necessary to compare three different pairs of models with each other.

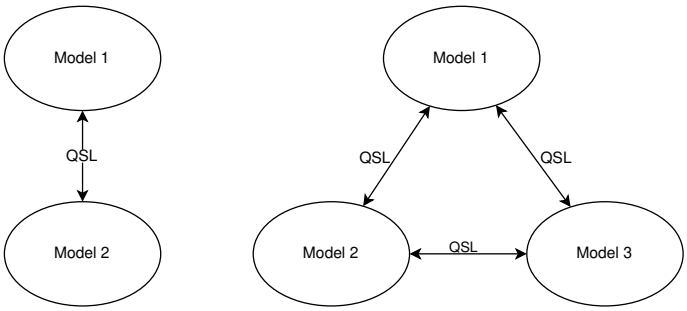

Figure 3.1: The increasing amount of different QSLs in larger model sets

Combining these individual levels therefore needs to follow an aggregation rule. For this reason, the possible options for calculating the QSL of a model set that have been designed are `min`, `max`, and `avg`. While with `min` the lowest level is chosen as the overall QSL for a model set, `max` chooses the highest level. `Avg` produces the mean value of all three measured levels.

A necessary addition to the QSLs was some form of normalization. As the value of the QSL of a model set will be combined with the Aggregated Model Score, both values have to be in the same interval in order to produce a meaningful Model Set Score that can be easily interpreted. It was therefore decided to introduce a QSL Score which is derived directly from the actual QSL. The level is put into the following formula which was derived and modified from Sünkel et. al. [11]: $\frac{(QSL \cdot 10)+60}{100}$. Essentially, the formula puts the QSL Score inside an interval between 0.6 (if the QSL is 0 - no query sharing is possible) and 1 (if the QSL is 4 – same window size and features). This interval was chosen to fit the overall values that are to be expected in the Aggregated Model Score.

The following summary is intended to give a better understanding of all the metrics and scores invented and used for the model set retrieval process. Figure 3.2 then shows how all those different scores are constructed and their relations to one another.

- **Performance Score**: Score for a single model. Chosen performance metric (e.g. accuracy, RMSE,...) is normalized over all models retrieved from the model database in a query. Assumes a value between 0 and 1.
- **Resource Awareness Score**: Score for a single model. The chosen resource awareness metric is normalized over all models retrieved from the model database. In the implemented version, this is a count of the used features of a model, with an additional penalty for small window sizes. Assumes a value between 0 and 1.
- **Model Score**: Overall score for single model. Combines Performance Score and Resource Awareness Score using a predefined weight.
- **Aggregated Model Score**: Score for model set. Determined by calculating the mean value of the Model Scores inside a set.

- **QSL**: Metric for model set. Depicts the degree of shared elements between two models inside a model set. Is determined for every 2-model combination inside a model set and subsequently ascertained for the entire model set by using one of three aggregation functions. Assumes value 0, 3, or 4 for each model-to-model relationship in a model set.
- **QSL Score**: Score for model set. Derived from the QSL by using the normalization formula $\frac{(QSL \cdot 10)+60}{100}$.
- **Model Set Score**: Overall score for model set. Combines Aggregated Model Score and QSL Score using a predefined weight.

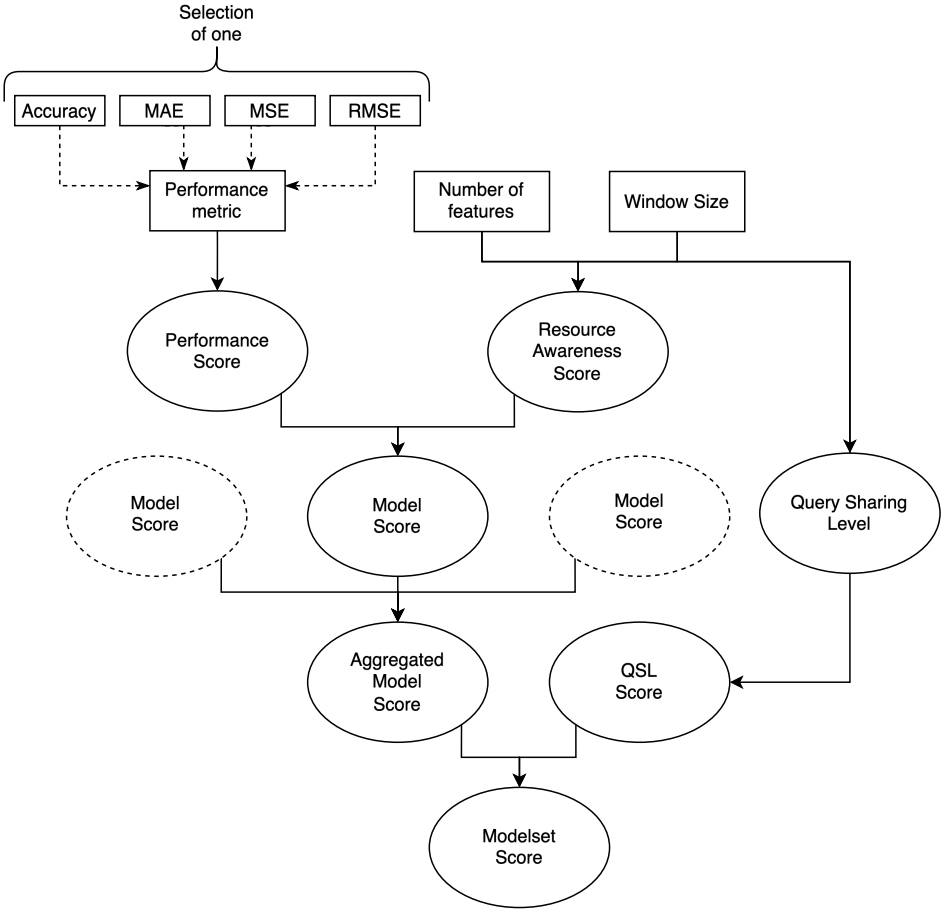

Figure 3.2: Structure of the different scores used for the retrieval process

As the model set retrieval system focuses on the logic of retrieving the k best model sets customized to the users' preferences manifested in numerous settings that have been made prior to the request, the design and implementation of a front end e.g. in the sense of a web app has been dispensed with. Whilst an easy-to-understand user interface that guides the user through the various functions of the model set retrieval system might lead to a more intuitive and time-saving engagement between the user and the system, this part was not the focus of this work and might be better suited for future development.

4 Implementation

The following chapter will give an overview of how both the latest version of the training pipeline as well as the model set retrieval system was implemented, how the user can customize the retrieval process by changing certain settings, and what challenges were faced during the implementation process.

4.1 Model Database

As mentioned in the previous chapter, there were many fundamental things to improve upon the already existing training-pipeline, when the development phase for this project was first started. The first step was to revise the model attributes by critically thinking about what information a machine learning model in the database should contain. As a result, the following attributes were added to the models in the database:

- `model_id`: Simple serial integer value that serves as primary key and makes it possible to refer to every created model in a precise way.
- `developer`: Attribute that makes it apparent, who trained and stored a specific model.
- `created_time`: Timestamp of when the model was successfully stored in the model database.
- `model_size_in_bytes`: Integer value of the memory size of that specific model in bytes. This value is calculated by returning the length of a byte array that is filled using a ByteArrayOutputStream on the content of the classifier.
- `training_weeks`: Shows how many weeks' worth of preprocessed training data was used to train a model. In the previous version of the training pipeline, one had to specify the amount of data entries (and therefore rows of raw parking data) that should be used for training. With the rework, the user now specifies the training data size in weeks instead. It is important to note, that for a fixed trainingWeeks value, a small window size value leads to more data entries used, than a big window size.
- `window_stride`: For the sake of completeness the window stride was added as a model attribute and depicts the time difference between the start points of two consecutive data entries used to train a model. In the present use case, the window stride is always the same as the window size.
- `start_of_training_data`: Depicts the month and year of the first entry of training data that was used for a model.
- `prediction_horizon`: The newly implemented prediction horizon of a model in minutes. Further information about this metric will follow in the next chapters.

In addition, the following attributes are reworked versions of previously implemented attributes. They were changed, either to be more comprehensible and uniform with the rest of the projects' terminology or to refactor the actual values to be better tailored to the implementations' functionalities.

- `model_name`: Instead of choosing an arbitrary name for a model, the model_name attribute was reworked to automatically give a model a standardized name based on its specifications, which makes it easy to extract important information without having to look up each metric separately. The syntax for the naming convention is as follows: `Classifier-Features-WindowSize-TrainingDataSize-ParkingLotID-PredictionHorizon-potentialSpecialProperty`. The features are separated by commas and are indexed as seen in Table 4.1.

Features	**Index**
temperature	0
humidity	1
weekday	2
month	3
year	4
time_slot	5
previous_occupancy	6

Table 4.1: Indexes of the features used for training

- `training_data_size`: Previously called `table_length`, this attribute states the total number of rows of preprocessed training data that were used for the training of this model. The value is calculated using the following formula: $training\_weeks \cdot 7 \cdot \frac{1440}{window\_size}$.
- `attributes`: Gives information about the attributes that were used to train this model. Instead of showing "all", if every possible attribute was used in training, in the overworked version now every single attribute is listed for better readability.
- `classifiers`: The same improvements that were underdone for attributes were applied to the classifiers attribute for better readability.
- `accuracy`, `mae`, `mse`, `rmse`: A models performance metrics. In the initial version of the training pipeline, these metrics were calculated and put into one single column, making it unwieldy to process further in the top-k algorithms (As an example, in previous versions of the top-k retrieval system, the accuracy values had to be extracted using a `re.search` function on the string that contained all the metrics, and then converted to a float value). By making the metrics atomic, they can be accessed more directly without wasting computational resources. Another advantage is the better readability for the user.
- `space_ids`: Previously called `slotids`. Indicates which parking spaces were considered for training. In the executed pipeline-runs, every parking space was selected.
- `model_content`: Byte array of the model. In the original training pipeline the content of the model was split into four different attributes, one per classifier. As there is only one classifier used per model, this was changed to always display the content in this attribute.

The following attributes were mostly adopted directly from the previous implementation, but will be introduced anyway to give context about their meaning:

- `parking_id`: The ID of the parking lot the model is trained on. In the current state, the parking lots 38 and 634 are available.
- `window_size`: The window size of the data entries used for training. This value describes how many minutes of parking activity should be summed up into one row of training data in the preprocessing phase. The models that were trained using a prediction horizon have a window size of 1 or 5.
- `trainingdataproportion`: Shows the proportion of the training data in relation to the test data. A value of 0.8 for example puts 80% of the data into training, and 20% into testing.
- `accuracypercent`: Metric that decides how many percent points the prediction of the model can deviate from the actual target value without being categorized as a wrong prediction.
- `randomforestmaxdepth`: Hyperparameter for Random Forest classifiers. The maximum number of splits each decision tree in the Random Forest can make.
- `kneighbours`: Hyperparameter for KNN classifiers. The number of neighbors of a data point that should be checked for classification.

The combination of these attributes made it straightforward to both filter for models when searching for something specific in the database, as well as process relevant information for the retrieval system.

4.2 Training Pipeline

With the structure of the model database explained more insight into the training pipeline can now be given. The functionality of the original model trainer was to create single models by specifying their properties in a `config.properties` file. After a new `Settings` object has been created that reads the `config.properties` file through an `InputStream`, a connection to the model database was established. Subsequently, the raw parking data was queried from the database and then preprocessed according to the specified properties. The preprocessed data was then used to build, test, and finally store the model in the model database. The process was then ended. It was apparent that simply implementing a repeating sequence of this process was not sufficient for a working training pipeline. Especially the recurring opening of new connections to the database was an obvious thing to be optimized. By restructuring the way the function `createDBConnection` is called, not only the problem of only establishing five connections at a time could be bypassed, but more prominently vast time savings could be accomplished.

The general functionality of the training pipeline is based on several nested for-loops, each referring to one metric that is to be changed. Therefore, one for-loop is for example changing a predefined variable that stores the window size that should be iterated over. In order to make an insight into, as well as possible changes to the values in an uncomplicated manner, most of them have been stored in `HashMaps`. A `HashMap` may store additional information as well, and so the `windowSizeMap` also contains the values of `trainingWeeks` that are attached to the window sizes. In order to provide the model trainer with those changing properties of each new model that was to be built, a new function `changeValues` was implemented. Using a `FileOutputStream`, the values of the iteration are filled into the properties file, which was renamed to `training.properties`. During several test runs of the model trainer, however, it was noticeable that the changes to the properties file

only came into force one iteration after the actual change. The reason for this was that changes in the `FileOutputStream` were only reloaded after each iteration. The `Settings` class therefore had to be modified in a way, so that it takes the already changed `Properties` object as an input parameter. The afterward created `ModelTrainer` object then directly refers to the changed values of the current iteration.

A first longer test run led to the fixing of some minor problems, one being the handling of double-training the same models: The original model trainer was built in a way that encoded the features a model should be trained on by assigning each feature to a number. Deselecting all features, let to the training of all of them. Although, by selecting every number, the same result was achieved. Such unexpected behavior was adjusted by modifying the for-loops. More importantly, however, another inefficiency was addressed in this development stage. As the models mainly used the same subset of training and testing data, the `preprocessing` function was operating on the same data each time a new model was created. It was then decided to outsource the preprocessing phase into its own class `Preprocessor` and define a new database table that would store the preprocessed data for it to then be accessed directly by the model trainer as needed. Essentially, a second pipeline was therefore created that would take care of preprocessing data in reference to certain properties that have been set beforehand. The general procedure in this new preprocessing pipeline is very similar to the way the training pipeline works: The user uses the `HashMaps` to set up the parking lots and the window sizes that should be preprocessed on. A `changeValues` function that is similar to the one used in the `ModelTrainer` then changes the values in a `preprocess.properties` file via an `FileOutputStream`. After that, the raw parking data is queried from the database and is then preprocessed by segmenting it into windows of the previously set size. The window size is the same as the window stride, leading to sliding, non-overlapping windows [12]. The preprocessed data containing parking, time, and weather information is then put into SQL statements which are finally executed every 1000 rows. By using a batch size of 1000, long waiting times and potential errors by creating very large statements can be avoided. In order to correctly select the required type of preprocessed data later in the training pipeline, based on the `pID` and window size of the model that is to be built, it was decided to include this meta-information in the table containing the preprocessed data as well. This was first done through an attribute called `context` that contained both `pID` and the window size but was later split up into atomic attributes to be in line with the norm of designing database management systems and to avoid inconstancies [13].

The attributes of the preprocessed data mostly represent the features the models can be trained on: As mentioned before this includes parking information (`previous_occupancy`, `occupancy`), weather information (`temperature`, `humidity`), and time information (`weekday`, `month`, `year`, `time_slot`). `Time_slot` has been called `timeHorizon` in the previous version of the model trainer and was therefore renamed in order to avoid misunderstandings. It describes the number of the "slot", a data entry refers to, by dividing up the 24 hours of a day into time frames of the same size as the windows used for preprocessing. For example, the time slot 2 for a window size of 5 would be the time window from 00:10 AM until 00:15 AM (as the encoding of the first time slot in a day starts with 0). Two other attributes were also added: `period_start_time`, which is a timestamp of the period it is referring to, and `shift24h`, which will be discussed in the next subchapter.

After a full run of the preprocessing pipeline, all available raw parking data is available and ready to be used for model training and testing for the desired window sizes, even if not every data point is actually used for model building (e.g. because of a small `training_weeks` value). Besides the vast

time and computational resource savings that are accomplished this way, the preprocessing pipeline also has another important advantage: With it, it is possible to select and pinpoint the exact number of rows of the preprocessed data and therefore the exact time span of the training data used, by utilizing `training_weeks`. Without the `Preprocessor` class, the only option to specify the amount of training data used was through a metric called `tableLength`, which described the number of rows of raw parking data to be used for preprocessing. Because a varying `pID` and window sizes lead to a different number of preprocessed data rows, however, this way of selecting the size of the training and test data is more imprecise. Having already established preprocessed data ready, makes it possible to know exactly how many data points were used in the training or testing of a certain model. Another addition that was implemented into the training pipeline was the use of feature scaling, mainly normalization and standardization. This was done using `Filters` of the Weka library. A normalize or standardize filter could simply be applied to the training- and test-dataset in order to scale the values in a preferred way. The normalization filter rescales all attributes to values between 0 and 1, while standardizing the data sets the mean value to 0 and the standard deviation to 1. The results of using feature scaling in the training process will be discussed in chapter 5.

4.3 24 Hour Shift

A central part of the implementation phase that led to numerous challenges and reconstructions was the so-called 24 hour-shift. It originated from the idea, to train models that would predict the parking occupancy for the same time slot but 24 hours in the future. Up until that point, models could only really "predict" the occupancy value for the time slot the model was run in. Let's take a model `dt-1,5,6-60-4-38` as an example. As described earlier, one can derive the most important information about the models' metrics from its name. In this example, the model works on a Decision Tree classifier, it was trained on the attributes humidity, `time_slot` and `previous_occupancy` using a window size of 60 minutes and four weeks' worth of training data. Furthermore, it predicts values for parking lot 38. Using the window size and the `training_weeks` value, it is evident that $(24 \cdot 7 \cdot 4 =)672$ instances were used for training. One of those instances may look like this:

humidity	time_slot	previous_occupancy	occupancy
71	14	63.7	59.3

Table 4.2: Exemplary instance for model `dt-1,5,6-60-4-38`

The first three columns are the features that are used to train the model, while `occupancy` is the target attribute. `Previous_occupancy` points to the occupancy value of the previous window (so in this case, time slot 13), while all other attributes display values of the current window. If this exemplary model was put into production it would therefore take the current humidity (for example by fetching live data from a nearby weather station), the current time slot (simply by knowing the current time), and the occupancy value from the previous time slot (by looking it up in a database for example), and then predict the occupancy value for the current time slot using those three attributes. For simplicity, let's assume that the current true values correspond to the values in the instance shown above. Being in time slot 14 (with a window size of 60, at the time of prediction it would therefore be between 14:00 and 15:00), the parking lot would be occupied for 59.3 % in this hour on average. The model now predicts how occupied the parking lot is on average in the current time slot. The benefit

of this prediction highly depends on the exact point in time when the prediction is being done: If the model predicts the occupancy value for time slot 14 at the beginning of said time slot (e.g. at 14:03) there certainly might be value to the prediction, as the majority of the current time slot has still not happened. However, if the prediction is done near the end of the current time slot (e.g. at 14:58) the informative value might be completely negligible: The prediction corresponds mainly to the past 58 minutes, and while it still might represent the actual current occupancy to some extent, can't really be considered a prediction in the proper sense.

During the implementation phase, it became apparent that this behavior was not aligned with the desired use case of the parking availability prediction system. While the models do predict in the sense of machine learning, the prediction cannot really be considered as such, as the use case requires something else: The predicted parking lot occupancy in a specific amount of time. It was then decided to implement the previously addressed 24 hour-shift. For this, a special type of preprocessed data first had to be stored using the preprocessing pipeline. Instead of using the time information of the current time slot as attributes, the time attributes of the previous day, but the same time slot (therefore having a time difference of 24 hours) would be looked up and put into the corresponding column of the instance. For this, the previously introduced attribute `period_start_time` was used: A new timestamp was created using the `minusHours(24)` function from the `Java.time` library. Subsequently, the `weekDay`, `month` and `year` attributes were adapted to this new timestamp. Note, that the `time_slot` attribute was not manipulated, as a shift of exactly 24 hours means the time slot stays the same within a day. Further, the `previous_occupancy` value was set to -1. The reason for this was, that speaking from a practical view, using `previous_occupancy` as a feature in production wouldn't be of much sense when using the 24 hour-shift: There was no way of knowing for certain, how high the occupancy was in the time slot before the time-to-be-predicted. It was therefore decided that `previous_occupancy` would be unavailable for all models trained on the 24 hour-shift. With the time information attributes pointing to a timestamp 24 hours before, the occupancy value staying the same and the previous occupancy value essentially being set to null, the weather information attributes were not altered. This also had a practical reason: It was reasoned that when using the 24 hour-shift models in production, the weather information of the point in time to be predicted could be gathered relatively easily by using a weather forecast. The limitations of this approach and other possibilities for handling the weather data will be discussed in chapter 6. Models that were trained using the 24 hour-shift, are marked with a `-24h` suffix in their `model_name attribute`. In concision, by implementing the 24 hour-shift, it was possible to have a model predict the occupancy value of a point in time in the future, rather than predicting only the current time slot.

Another addition to the training pipeline was the option to change hyperparameters for certain classifier algorithms. The goal was to both change the number of neighbors for the KNN algorithm, as well as the maximum depth of the Random Forest algorithm to introduce more diversity into the model database and identify potential changes in performance. While these hyperparameters could be changed manually in the original version of the `ModelTrainer`, an automatic altering of the values was introduced by implementing the function `changeHyperparameters` that works very similar to the previously introduced function `changeValues`, and is called by using another for-loop in the main method, that is skipped if a classifier that doesn't have any changeable hyperparameters (Decision Tree and Linear Regression) is being built. For storing the values of `kNeighbours` and `randomForestMaxDepth` new `HashMaps` were defined, just like for the other values that are being changed in the training pipeline.

4.4 Running the Pipeline

After all these changes were applied to the training pipeline, the first proper run of the training pipeline was carried out. Here, the aim was to create as many diverse models as possible, using the previously introduced attributes such as the window size or the training data size as settings to adjust. The created models can be accessed in the table `niklas_trained_models` which is located in the universities `postgreSQL` database. The following attributes have been changed systematically so one model with each possible combination was trained. Each metric's name is followed by the values that each attribute was given in the training run:

- `parkingLot`: 38 or 634.
- `windowSize`: 10, 30 or 60 minutes.
- `trainingWeeks`: (training data size in weeks worth): 1 or 4 weeks.
- `classifier`: Decision Tree, Random Forest, Linear Regression, KNN.
- `attributes`: `temp`, `humidity`, `weekday`, `month`, `year`, `previous_occupancy`.

The attribute `timeslot` was included in every built model. In addition, special models containing the following metrics were trained:

- **24 hour-shift**: Only for window size of 60 minutes; attribute `previous_occupancy` was always deselected in the case of a shift.
- **Feature Scaling (Normalization)**: Only for `windowSize` of 60 and `training_weeks` of 4.
- **Hyperparemeters**: For Random Forest and KNN classifiers, changes in hyperparameters were considered. Each Random Forest model exists in variants of `maxDepth` = 1, 5, and 20. Each KNN model exists in variants of `kNeighbors` = 3, 19 and 33.

While changing the following metrics is possible using the current version of the training pipeline, they were not altered in the training run. The metrics' name is followed by the value that has been chosen for this run:

- `trainTestStrategy`: Test after Train (using the first data points as training, the remaining for testing).
- `trainingDataProportion` (proportion of training data in relation to test data): 0,8.
- `accuracyPercent` (tolerance in percent for accuracy calculation): 1.
- `slotsIDs`: All parking slots were considered for training.

Using these combinations, a total of 4603 models have been trained, tested, and stored in about 8,5 hours. Models using the Random Forest algorithm used up the most time during building, with an estimated average of about 30 seconds per model. Observations regarding the performance of the generated models will follow in chapter 5.

4.5 Prediction Horizon

After the first proper model training pipeline has been successfully run, discussions about the true meaning of the metric `windowSize` arose. In various discussions and debates, it was thought about how to accomplish a proper parking lot occupancy prediction, without having to use the procedure for the 24 hour-shift introduced before. Further discourse showed that the original idea behind `windowSize`, was not only to segment the raw parking data into windows of this size, but also use it as a way of predicting future occupancy values. Furthermore, discussions lead to the realization that higher window sizes like 30 or 60 minutes might not lead to accurate results for predictions that lie in the future for only a short time, like 10 minutes. It was decided that the time difference between the point of prediction and the time-to-be-predicted should be called prediction horizon. Subsequently, the following conclusions were made:

1. Training models using a smaller window size (like 1 or 5 minutes) will lead to more fine-grained preprocessed data that might produce better results.
2. Using these models trained on small window sizes without any form of future prediction aspect will have high accuracy but very low informative value. (E.g. "predicting" the average occupancy for this very minute is not of value for the user: Current occupancy could easily be viewed through live data from sensors etc.)
3. Combining the small window sizes with some sort of shift might produce accurate and meaningful predictions. However, if the procedure for the existing 24 hour-shift is to be reused, many preprocess-runs must be made for each combination of window size (e.g. 1 and 5 minutes) and time horizon (e.g. 10, 30, and 60 minutes). This might make future training unflexible, as data first has to be preprocessed for every different prediction horizon and window size.

Following these conclusions, it was decided that by utilizing preprocessed data segmented on small window sizes and applying a prediction shift after the preprocessing, an agreement between accuracy, meaningful results, and flexibility could be met. Therefore, a new variable called `predictionHorizon` was introduced to the implementation. Therefore, the time difference between the time of prediction and the time of `occupancy` to be predicted could be represented much clearer and more flexibly. The values that `predictionHorizon` is to adopt, are stored in `HashMaps`, just like for the other variables. As the preprocessing is not done specifically for every different prediction horizon beforehand, more data than the actual specified data size for training and testing - as set in `trainingdata_size` - has to be gathered from the database. In total, the number of data points required for training and testing equates `training_data_size` + `predictionHorizon`.

The reason for this is the way the data instances for training and testing are created: While most values stay the same (i.e. weather and time information), the `previousOccupancy` is set to the `occupancy` value. Then, the `occupancy` value is set to the `occupancy` value from the next data instance. The handling of the training and testing data in this updated version is therefore quite different from the data handling in the 24 hour-shift. Rather than discarding the `previousOccupancy` attribute, it is now used to correspond to the `occupancy` value of the current time slot. The target attribute now is set to the occupancy value in `predictionHorizon` minutes. This value is fetched by looking at the index of the current row of the training/test data and adding the `predictionHorizon` value divided by the window size value by it. For a better understanding, the actual implementation for handling the prediction horizon in code is shown in Figure 4.1.

```
// Instance handling for prediction horizon
// Prev. Occ set to current Occ; Occ set to the value in predHor.-Minutes
// Other values stay the same
for (int i = 0; i < trainingDataSize; i++) {
  res.row(i).setDouble("previousOccupancy",
      res.row(i).getDouble("occupancy"));
  res.row(i).setDouble("occupancy",
      res.row(i+(settings.predictionHorizon/settings.windowSize))
          .getDouble("occupancy"));
}
```

Figure 4.1: Modificaiton of training data regarding the prediction horizon

Utilizing this newly created way of referencing the time that is to be predicted, a second large run of the training pipeline has been executed. The changing metrics in this run stayed the same as for the first run with the following exceptions: Following conclusion 1 from above, the models of this run were trained only using window sizes of 1 or 5 minutes. Additionally, all models were trained with the newly introduced variable `predictionHorizon` being either 10, 30 or 60 minutes long. A short summary regarding the performance of the models created in the second run of the training-pipeline will also follow in the evaluation chapter.

4.6 Retrieval System

The general aim of implementing the model set retrieval system was to provide the user with a highly customizable option to retrieve models and model sets that are tailored to their needs and the specific use case, whilst relying on different algorithms that have been well established in prior scientific work. This subchapter aims to give an overview on the functionalities of the retrieval system as well as the development process.

After setting up the development environment, the first functions that were implemented were simple READ- and CREATE endpoints called `select_direct` and `select` that were introduced before. As a next step, the actual retrieval system was to be implemented. To create a basic working function, it was decided to first focus on retrieving only single models instead of model sets, as to that point of time in the development phase there has not been made a choice on how to approach model set creation. This first created retrieval function is called `topk` and starts by reading and storing all the user's inputs. Subsequently, the environment variable containing the URL and login information for the `postgreSQL` database is loaded and a corresponding connection to the model database is being established. Then, a SQL statement containing information about the relevant machine learning models is prepared and executed. Once the models are obtained from the model database and stored in the `Table` datatype, each model is prepared so both the performance as well as the resource awareness metric are easily accessible, storing the corresponding values in columns called `1` and `2`. These generic names were chosen on purpose in order to make the processing of other data easier without having to change anything in the code itself. The models are then further curated by normalizing the metrics, creating tables each containing only one metric, sorting them, and storing these tables in a `dictionary` datatype. At this point, the lists of models are ready to be further processed by the top-k algorithms. Therefore, in the next step, the algorithm that was selected by the user is run,

producing a list of the top-k models, that are then converted into JSON together with relevant meta information.

Before going into detail on how the functions for model set retrieval were implemented, Table 4.3 will give an overview of what settings the user can make to customize their retrieval request. The description of the settings is followed by the possible values the settings can take on in the current implementation.

The implementation for the model set retrieval is largely similar to the `topk` function up to the point of starting the top k algorithms. As model set retrieval requires multiple different prediction horizons inside a single request, the selected algorithm has to be run separately for each prediction horizon. For instance, FA chooses the best models with a prediction horizon of 10, 30, and 60 minutes one by one and puts each list into another list named `result`. To follow the idea of creating actual sets of models out of those selected single models, the helper function `create_combinations` is called. This function then generates all possible combinations of models, using one model per prediction horizon. If `combineSameFeatures` is set to true in the API call, `create_combinations` only creates sets of models that use the same features. At the time of implementation, this was the only way to ensure some sort of inter-model resource awareness, which was later complimented by the QSLs. The implications of the `combineSameFeatures` setting in regard to the QSLs will be discussed in chapter 6.

After all possible model combinations have been created, the sets are then further prepared for the second algorithm call. For that, a new `dictionary` containing both the model sets themselves as well as meta-information like identifications and scores is created. The relevant scores in this step are the Aggregated Model Score and the QSL Score which are ascertained or calculated here. The determination of the QSLs follows the user's settings for the QSL aggregation rule. After the relevant scores are established, the model sets are again split into two lists, one for each scoring metric. The labels for the two lists are `1` and `2` again to make the processing in the top k algorithms clearer, however, the underlying metrics are now the Aggregated Model Score and the QSL Score. After each list has been sorted and the specified top-k algorithm is run again, the result is converted into a JSON readable format and finally returned to the user.

To run the top-k retrieval system, the user must first start the Flask app whilst being connected to the university's network (directly or by VPN). To do that, the user has to change into the directory `topkretrieval`. Then, the virtual environment for the implemented system must be entered by executing `source .venv/bin/activate`. Afterwards, the app can be started by running `flask run -port 8000`, or any other arbitrary port. The retrieval system can then be accessed by using an API client like Insomnia and reaching the endpoints stated above. For instance, to reach the endpoint for model set retrieval, a POST statement would have to be sent to `http://127.0.0.1:8000/api/topk/modelsets` appending the necessary JSON data explained above.

In summary, both the preprocessing- and training-pipeline as well as the top-k retrieval system have been designed and implemented in a way that makes it possible for future users to simply preprocess, train, and retrieve models and model sets without having to do any major changes to the code. The implemented modules are ready to use once the desired settings and metrics are put in the pipelines corresponding `HashMaps` or the CRUD statement of the API endpoint one is calling.

Setting	Description	Values
pID	The parking lot ID that the model set should predict for	`38` or `634`
windowSize	The window size option(s) of the models inside the model set	`1`, `5` or both
perfMetric	The performance metric to be used to calculate the performance score	`acc`, `mae`, `mse`, `rmse`
k1	Number of models to consider for model set creation per prediction horizon. E.g., if k1 = 3, for every prediction horizon the top 3 models will be chosen to then further create all possible combinations of model sets. When having two prediction horizons, this means $3 \cdot 3 = 9$ different model sets are created	`1` to `n`, where `n` is the maximum number of models found in the model database
k2	Number of model sets to be returned to the user. Out of all created model sets in the first top-k round, the top k2 are then selected. Setting k2 to `max` will automatically return all created model sets. E.g., when having three different prediction horizons, `max` will return $3 \cdot 3 \cdot 3 = 27$ different model sets	`max` or `1` to `n`, where `n` is the maximum number of model sets that can be created using k1 models
predHor	The different prediction horizons to be considered in the model sets	`10`, `30`, `60` or a combination of those
perfWeight	The weight of the Performance Score in relation to the Resource Awareness Score. E.g. a value of 0.8 will compute the Model Score using 80% of the Performance Score and 20% of the Resource Awareness Score	Value between `0` and `1`
AMSWeight	The weight of the Aggregated Model Score in relation to the QSL Score. E.g. a value of 0.8 will compute the Model Set Score using 80% of the Aggregated Model Score and 20% of the QSL Score	Value between `0` and 1
algorithm	The top-k algorithm to assess the top k items for both rounds	`naive`, `fagin` or `threshold`
combineSameFeatures	Option that makes it possible to only create model sets where all models use the same features	`true` or `false`
calculateQSL	Aggregation function for determining the overall QSL of a model set	`min`, `max` or `avg`

Table 4.3: Settings in the Retrieval System

5 Evaluation

The evaluation of this work is divided into two parts. First, there will be a short summary of the observations that could be made after the previously mentioned run of the training-pipeline was executed. In the second subchapter, the implemented top-k retrieval system will be evaluated using different methods. Here, the effects of the integrated top-k algorithms will be examined first and compared to one another. By doing that, both the number of accesses to the sorted lists as introduced in chapter 2, as well as the runtime of the algorithms, will be analyzed. Afterward, an evaluation of the impacts of resource aware model sets on the data usage will be made.

5.1 Training Pipeline

Upon execution of the training-pipeline runs that were introduced in the previous chapter, several different observations could be made. The following analysis of the models that were created in the previously mentioned training pipeline runs will be done for all created models – models that were created in the first as well as in the second run. Therefore, observations about models with window sizes of 1, 5, 10, 30, and 60 minutes will be made. Additionally, the performance metrics of models that use the concept of prediction horizons will be analyzed together with models that use the previously introduced 24 hour-shift or no future prediction aspect at all.

One thing that immediately became apparent when looking at the created models, was that small variations in certain metrics could make big differences in performance. One of the biggest influences comes from the window size of a model: a small segmentation mostly means an increase in performance metrics like accuracy, compared to a large segmentation. Small window sizes for models using a prediction horizon are always performing better, than big window sizes for models using a prediction horizon. This observation holds for models using the 24 hour-shift and models with no future prediction aspects. However, the accuracy for models with neither prediction horizon nor 24 hour-shift tends to be higher than for models that use some form of future prediction: In total, 250 models that use no form of future prediction have an accuracy value of over 90%, while the same is true for only 116 models that make use of a prediction horizon. The highest accuracy value for a model that was trained using a 24 hour-shift only reached an accuracy of 67%.

Another observation that was made, is that feature scaling seemed to have close to no impact on the performance of the models. The vast majority of performance metrics (accuracy, MAE, MSE, RMSE) stay the same for a model compared to its feature-scaled pendant. However, as stated in the previous chapter, feature scaling was only done for some models. Further tests and evaluations using feature scaling on models with different window sizes and prediction horizons could produce unexpected new insights.

The feature `previousOccuapncy` seems to be the feature with the most impact on performance. Out of all trained and stored models with an accuracy of over 90%, 346 use `previousOccupancy`, and only 20 don't. However, 18 out of those 20 models allegedly have an accuracy of 100%, which with a high probability makes them candidates of overfitting. This speaks for an indispensability of using the occupancy of the previous time slot as a feature when training a new model. Models that don't have access to this information seem to perform worse overall. When looking at this finding from a theoretical perspective, it should not be hard to see the reason for this behavior: In between two consecutive time slots (may it be one minute, or even 60 minutes), the occupancy of a specific

parking lot usually should not change too much. Of course, a considerable change in occupancy during rush hours in the early morning or afternoon hours would be not surprising. However, this volatility in occupied parking spaces would most likely not happen rapidly in a matter of a few minutes, but rather in longer time periods, like a few hours. It therefore does not come as a surprise, that `previousOccupancy` plays such an important role in a model's performance.

In summary, the observations that could be made when looking at the performance metrics of the models trained by the constructed pipeline are mostly in line with prior expectations. Window size and the feature `previousOccupancy` rightfully have a large influence on a model's performance. The lack of impact of feature-scaled models is something to further research and analyze.

5.2 Retrieval System

To evaluate whether the previously introduced top-k algorithms perform any better than a simple-to-implement naïve algorithm, various tests have been performed to show both time consumption as well as sorted and random accesses of all implemented top-k algorithms. In order to assess the necessary data about time consumption and computational resources used, the implementation of the algorithm has been changed slightly. Counters were added that would increment every time a sorted access or a random access was made. In addition, the Python standard module `time` was used to create time stamps at the beginning and end of each operating algorithm in order to exactly measure the time, each algorithm execution takes. Both accesses and time consumption have been measured separately for every algorithm round. Consequently, if an API call contains three different prediction horizons, in total four execution times are measured: One for each prediction horizon and one for the second round of top-k algorithms which selects the best model sets out of the created combinations.

To give insight into how the data for the evaluation was gathered, the used settings for the API calls are displayed along with the corresponding observations.

5.2.1 Accesses

Before comparing the number of accesses across the different kinds of top-k algorithms, first, it had to be made sure that inside each algorithm the number of accesses stayed the same, which was previously not the case. The reason for this was that the lists that were handed over to the top-k algorithm were sorted in an arbitrary way when dealing with the same values. For example, all models having 3 features were sorted differently in the list (while of course still being ranked after the models with 2 features, and before the models with 4 features). This results in varying favorability of processing which leads to different orders in each API call. Therefore, a second attribute has been added to the sorting function that is called when creating the different lists before executing the algorithms. It is therefore ensured, that when carrying out the same API call multiple times, all items contained in the lists (models in round one, or model sets in round two) are always in the exact same order, even when having to deal with ties. To minimize this randomness and have the models be sorted in the exact same way, the model ID has been added as a second sorting attribute. Subsequently, per algorithm the number of accesses stays the same for a predetermined request.

After the consistency of the number of accesses for an algorithm has been ensured, the actual evaluation could be started. To evaluate the number of accesses for each algorithm, three different

	Request 1	Request 2	Request 3
pID	38	38	634
windowSize	[1,5]	[1,5]	[1]
perfMetric	acc	rmse	acc
k1	8	6	10
k2	3	max	10
predHor	[10,30,60]	[10,60]	[10,30]
perfWeight	0.8	0.78	1
AMSWeight	0.65	0.2	1
algorithm	NA, FA and TA for every request		
combineSameFeatures	false	false	false
calculateQSL	min	avg	max

Table 5.1: Requests used for evaluating number of accesses

exemplary requests have been created, as seen in Table 5.1. The API calls were designed to represent a diverse spectrum of possible requests. Each request was then run for every algorithm each. As stated before, following Fagin's reasoning, TA will in every case require less than or the same amount of sorted accesses as FA [14]. This is indeed backed by the observed data: In every one of the three requests, TA required the least number of sorted accesses, followed by FA and NA. Across all algorithm rounds and requests, TA used a mean of 48,1 sorted accesses per algorithm call whilst this value was 132,7 for FA and 10222,4 for NA. The computation advantage of FA and TA over naively iterating through the entire model/model set list is apparent. This observation however lacks informative value, as NA does not do any random accesses, that the other two algorithms heavily rely on. It might therefore be necessary to look at the number of total accesses per algorithm, by summing up both random and serial accesses. Doing this, TA uses a mean of 146,2 total accesses per algorithm call, followed by FA with 379,1 and NA with still 10222,4. Therefore it can be seen that if even every form of access is considered, NA will still have significantly more accesses to do, in order to return the top-k results.

Regarding the number of random accesses, a comparison between TA and FA can be made. As TA in its design relies more heavily on random accesses than FA, the former was expected to have at least a similar number of random accesses than the latter. This is because, in TA, every accessed grade is instantly followed up by the randomly accessed missing grade of the same item. In FA, however, the seen objects are only randomly accessed at the end, when k objects have been fully seen already. However, the average number of random accesses of 98,1 for TA and 246,4 for FA, came as a surprise. Apparently, TA still uses fewer random accesses than FA, which speaks for TA using less computational resources. Only in one occurrence did FA use fewer random accesses than TA: Getting the top k model sets in the second request required not a single random access by FA, whilst requiring 44 by TA. This however is easily explained when looking at the specified k2 in this request. Selecting "max" as k2 results in an algorithm returning all items that are in the model set list. In the case of FA, this means the algorithm serial accesses every item in order and by the time the random

access phase is started, the algorithm already has seen every object in its entirety. FA does therefore not have to do any random accesses.

In summary, the expectations of both FA and TA have been fulfilled, as both top-k algorithms use significantly less accesses than a naïve approach of selecting the top-k elements. In addition, TA required fewer accesses than FA, making it the most access-friendly out of all three observed algorithms.

5.2.2 Time

After having assessed the number of accesses each algorithm has to do in order to return the top-k elements, the focus will now be the execution time of these algorithms. Here, it is important to not confuse the execution time of the model set retrieval process with the inference time of the models. The latter refers to the time a model takes between the model receiving an input and producing an output [15]. The execution or run time that will be analyzed in this chapter, however, addresses the aggregated time that passes between the calling of the algorithm function and returning its' result for every algorithm call.

Because of changes in local memory like cache and other simultaneous running programs, it is apparent that the execution time for each algorithm will at least slightly vary for each run. It is therefore necessary to run the respective requests for each algorithm multiple times and then take the average out of the observed execution times in order to make a fair and valid comparison. When undergoing the first runs, using the same requests that were used for assessing the number of accesses, it became evident that there was little to no observable difference in run time between the three different algorithms. Using the previously used Request 1 as an example, the difference in total execution time over all rounds (summing up the three prediction horizon rounds and the model set round) between the algorithms were not larger than 0,02 seconds. Time differences like this are too small to make any meaningful interpretations about the algorithms' runtime performance. To magnify the time differences and produce significant evaluation results it was then decided to add a much large number of models to the model database so the top-k algorithms would have longer lists to work with. Using a for-loop in `postgreSQL`, approximately 200.000 instances of the same model were added to the model database. For the evaluation, it was sufficient to only add the metadata about the model, like window size and features, without having to add the actual model content as a byte array. Table 5.2 shows the subsequently created request that made the algorithms consider those newly added models.

As expected, the differences in runtime became apparent once a larger model database was used. The request shown in Table 5.2 was run 20 times per algorithm, each time adding up all the individual times per algorithm run (in this case 2 prediction horizon runs plus one model set run). As an average, it took NA 4,464 seconds to come up with the top-k results, whilst TA and FA took significantly less time: On average, FA returned the top-k objects in 0,032 seconds total, while TA took 0,030 seconds. While the difference in runtime of FA and TA is hardly interpretable, the longer runtime of NA is explainable by its general procedure: Having to access every item in a list of over 200.000 items takes up much more time than only accessing the topmost items in each list.

Nevertheless, it is important to mention that all stated times in this chapter refer to the net time the program spends in the specific algorithm function. The whole requests itself take up more time, especially when the additional added 200.000 models come into consideration. In this case, it is not uncommon for the whole request to take over 60 seconds. This can be traced back to the overhead of

	Request 1
pID	634
windowSize	[5]
perfMetric	acc
k1	3
k2	5
predHor	[30,60]
perfWeight	0.75
AMSWeight	0.2
algorithm	NA, FA and TA
combineSameFeatures	false
calculateQSL	min

Table 5.2: Requests used for evaluating number of accesses

the model set retrieval system, mainly the preparation steps the lists must go through before they can be handed over to the specific top-k algorithm. Fetching the required data from the model database as well as normalizing and sorting it, will take up a large portion of time that makes the time spent in the top-k algorithm seem relatively short. It could therefore be argued that the additional 4 seconds that NA takes does not vehemently increase the perceived time from the request sent to receiving the returned response. Nonetheless, the measured times show the advantage of top-k algorithms that make use of random accessing: Not having to access every item serially does save time and computational resources. Even though the main app of the retrieval system offers several options for optimization in regard to runtime, TA and FA both offer a direct way to save execution time.

As TA uses significantly less sorted and random accesses than FA, it was expected to have a shorter runtime than FA. However, the similarities in runtime of the two algorithms can be caused by a variety of reasons. For one, an algorithmic overhead can cause TA to run slightly slower than expected. This of course could be due to varying processes running in the background of the local machine. In addition, sections like calculating the threshold value of TA could potentially cause to add time to the execution that could then further be compensated by the saved time accomplished by the fewer number of accesses. Furthermore, the chosen data structures like `pandas dataframe` could very well be not the most efficient way to handle the processed data. It is possible that other data structures like lists could make it possible to save some more fractions of a hundredth of a second in execution time. However, it is not the aim of this work to micro-optimize the provided source code. The evaluation of the implemented model set retrieval system shows that regarding execution time of the top-k algorithm, there is in fact a difference between a naïve approach and approaches that make use of random access. While these time differences might still seem marginal in relation to the total execution time, they might gain relevance once multiple queries are started after one another, or even a larger model database than the one utilized for evaluation is being used.

5.2.3 Network Utilization

An analysis of the implications of using resource aware model sets will finalize the evaluation of this project. Up to this point, the various options for creating model sets that make efficient resource utilization possible have been introduced and the decisions that lead to the current implementation have been explained. However, the final advantages of using resource aware model sets are yet to be properly seen. This section aims to give an overview of the benefits resource aware model sets can bring regarding data transmission. For this, a simple extrapolation will be made using the structure of the parking occupancy prediction use case as an example.

To show what implications the use of resource aware model sets has, there are some assumptions to be made. It is assumed that in a running prediction system, if m is the window size of a specific model, after m minutes this model produces a new prediction. This means that not only does a model with a window size of 5 minutes segment the training data into slots of the size of 5 minutes, but also does this model predict the parking occupancy for the chosen prediction horizon every 5 minutes. This behavior is also the reason for penalizing models with a small window size by reducing their intra-model resource awareness score: A small window size means more predictions which in turn require repetitious data transmissions. Another assumption is that features with more or less constant values like `year` or `month` have to be transmitted from the prediction system to the working model set for every prediction. In other words, there is no way for the model to directly access any time-specific features in order to save data transmission. Lastly, it is assumed that the datatypes and therefore the memory size of every feature is the same as they are stored in the preprocessed database. The reason for this presumption is, that it is likely that in a real-life scenario, accessing live data like temperature or humidity will require not only more time than getting other values but also more memory size than the single double value both temperature and humidity are stored as. For simplicity, this evaluation therefore assumes a predictable memory size for each feature. Table 5.3 shows the data types used to store each feature and the resulting memory sizes.

Feature	**Data type**	**Memory size**
temperature	double	8 bytes
humidity	double	8 bytes
weekday	integer	4 bytes
month	integer	4 bytes
year	integer	4 bytes
timeslot	integer	4 bytes
previous_occupancy	double	8 bytes

Table 5.3: Data types and memory sizes of the different features

For this evaluation, three different hypothetical scenarios will be looked at. This way, the implications of resource aware model sets in different circumstances will become apparent. For each scenario, two different explanatory model sets will be introduced, one having a low extent of resource awareness (Model Set 1), and the other one with a high resource awareness (Model Set 2). The meaning of the indexed features can be looked up in Table 4.1.

	Model Set 1		Model Set 2	
	Model 1	**Model 2**	**Model 1**	**Model 2**
Features	0, 1, 2, 3, 4, 5, 6	0, 1, 2, 3, 5, 6	5	5
Required bytes	40	36	4	4
Window Size	1 min	5 min	5 min	5 min
Bytes per minute	40	7,2	0,8	0,8
QSL	0		4	
Unique features	7		1	
Total bytes/min	**40**		**0,8**	

Table 5.4: Model sets for extreme example

Extreme Example

In this extreme scenario, that uses both extreme ends of resource awareness, the theoretical impacts on the data transmission become obvious. Firstly, the focus will be on the low resource aware model set. Model 1 uses all 7 features, resulting in a data transmission of 40 bytes per prediction, which is the maximal amount in the implemented system. Pairing this with a relatively small window size of 1 minute implies a data transmission of 40 bytes every minute the prediction is running. Regarding RA, this is the worst possible combination of metrics in the context of this work. Because of this, it does not matter how many features the second model of the model set utilizes: There are no additional bytes that could be added to the maximum data transmission of 40 bytes per minute. Contrarily, combining Model 1 with a highly resource aware model inside one model set, would not enhance the necessary data transmission either: All 40 bytes still have to be transmitted every minute, no matter how well the second model utilizes its resources. In other words, the damage done by Model 1 in this example is too severe to be alleviated by combining it with resource aware models inside a model set.

The highly resource aware model set in this example shows the other side of extremes: Two hugely resource aware models in the same model set, both using the same one feature (of the size of only 4 bytes, instead of 8 bytes) and the same relatively big window size of 5 minutes. These model settings are the best possible regarding resource awareness. In addition, the selected feature of Model 2 is also not changing the amount of data transmission of Model 1, just like for the low resource aware model set in this example. This would not be the case however if Model 2 would use other features than Model 1, which will be discussed in the next example. Because Model 1 and Model 2 in this highly resource aware model set are completely congruent, they require just as much data transmission as if there was only one model. This model set therefore only transmits 0,8 bytes per minute, which again is the least amount of data possible. While in practice it is unlikely that a comparison between the least and the most resource aware model sets becomes plausible, this example still shows the bandwidth in which the data transmission is settled in, depending on the utilization of resources.

Realistic Example

Next, a more realistic example will be examined. Here, the two observed model sets do not differ as much from each other as in the last example. However, the impact of resource awareness will still be made evident.

	Model Set 1		Model Set 2	
	Model 1	**Model 2**	**Model 1**	**Model 2**
Features	0, 2, 3, 5, 6	0, 1, 2, 4, 5, 6	5, 6	5, 6
Required bytes	28	32	12	12
Window Size	1 min	5 min	5 min	5 min
Bytes per minute	28	6,4	0,8	0,8
QSL	0		4	
Unique features	7		2	
Total bytes/min	**30,4**		**2,4**	

Table 5.5: Model sets for realistic example

The model set with the low resource awareness consists of two models, one using 5 features, and the other one using 6 features, out of which 4 are congruent to the one the former is using. This will result in an additional 12 bytes of necessary data transmission since 2 of the features Model 2 uses are not already used by Model 1. However, as Model 2 works on a segmentation of 5-minute windows, those additional 12 bytes only need to be sent every 5th prediction of Model 1. Because of the inconsistencies in window size and feature set, the QSL of the low resource aware model set is 0. Overall, this results in a total data transmission of 30,4 bytes per minute. On the other side, the models of the highly resource aware model set use the same segmentation as well as the same set of features, therefore reaching QSL 5. As the only used features are `time_slot` and `previous_occupancy` (which incidentally corresponds to a powerful combination when retrieving models that both have a good performance as well as RA), the transmitted data size is 12 bytes per prediction, which results in 2,4 bytes per minute when taking the 5-minute segmentation into account.

This example shows, that even in a less extreme scenario, the amount of data needed to be sent can be well over 12 times as big when working with a low resource aware model set in comparison to a model set that utilized its resources efficiently.

Large Model Set Example

In the following exemplary scenario, a model set containing three models (therefore predicting three different time horizons) will be compared to a model set of two models.

	Model Set 1		Model Set 2		
	Model 1	**Model 2**	**Model 1**	**Model 2**	**Model 3**
Features	5, 6	1, 3	2, 5, 6	2, 5, 6	2, 5, 6
Required bytes	12	8	16	16	16
Window Size	1 min	1 min	1 min	1 min	1 min
Bytes per minute	12	8	16	16	16
QSL	3		4		
Unique features	4		3		
Total bytes/min	**20**		**16**		

Table 5.6: Model sets for large model set example

This example aims to show the significance of assessing the QSLs inside a model set. As Model Set 2 consists of one more model than Model Set 1, it would be natural to assume it automatically required more data transference. However, as the feature set across all models of model set 2 stays the same, the resources to be transmitted can be shared, which means the same data does not have to be processed more than once. Even though the models of model set 1 use the same segmentation (which results in a QSL of 3), the non-consistent feature set of the models greatly increases the total needed data usage. It is therefore crucial to assess the QSLs when comparing two model sets, as shared resources might have a bigger impact on resource awareness than the number of models inside a model set.

QSL Counterexample

This last example serves as proof of the importance of intra-model resource awareness. For this, the two model sets from Table 5.7 are assumed.

	Model Set 1		**Model Set 2**	
	Model 1	**Model 2**	**Model 1**	**Model 2**
Features	2, 5	6	0, 2, 5, 6	0, 2, 5, 6
Required bytes	8	8	24	24
Window Size	1 min	5 min	1 min	1 min
Bytes per minute	8	1,6	24	24
QSL	0		4	
Unique features	3		4	
Total bytes/min	**9,6**		**24**	

Table 5.7: Model sets for QSL counterexample

When looking at the QSLs of both model sets, the first model set would be expected to have a higher data transfer than the second one. After all, the models of model set 1 do not only have two completely different feature sets, but also different segmentations. Model set 2 on the other hand uses the same feature set as well as a uniform window size of 1 minute across all models. Nevertheless, the transmitted data of model set 2 is more than double the size of the bytes per minute of model set 1. The reason for this behavior is easy to recognize when looking at the actual chosen feature sets and window sizes: By making use of more features (some of which even require 8 bytes to be sent), the base data transfer of model set 2 is multiple times bigger than for model set 1. Even if there are no additional costs for model set 2, as it's Model 2 utilizes the same features as Model 1, the data needed to be sent is still more than for model set 1. Another benefit of model set 1 is that even though there are additional costs for model 2, these only occur every 5th prediction of Model 1. The resulting 9,6 bytes per minute against 24 bytes per minute might therefore seem unintuitive when comparing the low QSL of model set 1 with the high QSL of model set 2.

This example shows clearly, that a high QSL does not necessarily mean fewer data transmissions need to be undergone. It becomes clear, that if models have a high feature count, a small window size, and/or use features with large datatypes, a high QSL does not help to reduce the already high costs of data transmission. Moreover, this shows the importance of addressing the previously introduced intra-model resource awareness. Looking at a model's window size and the number of chosen features is

a required prerequisite when assessing any score of a model set regarding resource utilization. While the concept of QSL is suitable for finding out whether or not some resources can be shared among models, it is often not sufficient enough to evaluate a model set's resource awareness.

In summary, these examples of constructed model sets show a variety of things to consider when assessing resource awareness: For one, the difference in data transmission can be huge. In the implemented system, the amount of data needed to be transferred per minute can be up to 50 times bigger for a model set with extremely low resource utilization, compared to a very resource aware model set. Even in less extreme circumstances, the amount of additional data usage might quickly add up over the course of several minutes or hours spent predicting future parking occupancy values. It has also been shown that the dimension of QSLs has to be considered when comparing model sets, even if they contain a different number of models. A model set comprised of three models does not automatically have a higher data usage than a model set of two models, provided that it uses its resources efficiently by sharing the same feature set. However, as shown in the last example, a high QSL does not subsequently imply a smaller data transmission. Measuring intra-model resource awareness is an essential requirement when the degree of efficient resource utilization in a model set is to be determined. Lastly, the evaluation has shown that introducing additional penalties regarding a model's resource awareness score for using relatively large features (i.e. temperature, humidity, and previous occupancy use up to 8 bytes instead of 4 bytes) could make sense.

6 Future Work and Conclusion

This work made various contributions to the model training as well as the model set retrieval process. These contributions range from enhancing already existent modules, for example by separating the preprocessing process for model training from the actual training pipeline, to designing and implementing new ideas, for example by introducing intra-model resource awareness as a counterpart for QSLs. The impacts of the constructed retrieval system along with its top-k algorithms have been evaluated on multiple metrics to show their benefits regarding the model set retrieval process, which met the scope of this work. Still, there are many aspects surrounding this work that should be addressed in future works in order to apply changes to the current system, evaluate different alternatives and design choices regarding the architecture of this project, and ultimately improve the current implementation of both model training pipeline and model set retrieval system. This chapter aims to both refer to areas that can be used for further work on the topics of this work, as well as to summarize the undertakings and impacts of this thesis.

6.1 Future Work

The entire assessment of both performance and resource awareness of models and model sets in this work has been done in the context of the model training and model selection process. Even in analyzing the number of accesses per top-k algorithm, the evaluation still takes place "before" any model is run in production, i.e. outputting any predictions. For a completely comprehensive evaluation of any model, however, it is crucial to understand its performance inside a complete and working environment. Therefore, future work should aim to locate and assess suitable metrics that refer to the models' ability to produce valuable predictions. One possible metric could be the inference time of a model. Evaluating, how much time a model takes to generate an output is a relevant step, especially when working with small window sizes and prediction horizons. Assuming a certain model is supposed to produce a prediction every minute, requires the model to take significantly less time than a minute to produce said prediction, otherwise its informative value would be of no use. This additionally applies to when working with small prediction horizons. This work produced model sets with the smallest prediction horizon being 10 minutes. If potential future work includes much smaller prediction horizons, however, e.g. 2 minutes, a long inference time would significantly reduce the usefulness of a model. For these reasons, analyzing the inference time of models and integrating it into its scoring function when in the process of selecting models for model set creation, might be a valuable topic to work on in the future. An adjoining question to further pursue would also be, whether to account the inference time of a model toward its performance or resource awareness score.

A metric that would definitely be valuable to contribute to a model's resource awareness score, however, is RAM usage. Potential follow-up work could integrate the number of bytes of RAM the training, storing, selecting, and in particular the inference of a certain model requires. Less RAM used up by one model means more available computational resources for other processes. Consequently, the demand for RAM of models correlates directly with its resource awareness. While RAM usage might already be indirectly addressed in this work when evaluating the number of accesses of the different top-k algorithms, the focus was not on the model itself but instead on the model set retrieval system itself. Also, the number of sorted or random accesses is probably not a reliable indication of RAM usage even though the two metrics might correlate. Thereby, a precise analysis of RAM usage might contribute to a more omniscient assessment of resource awareness as a whole.

As mentioned before, when working with a very large model database, the execution time the respective top-k algorithm takes up is relatively small when compared to the time the entire request takes. To make even complex requests faster, an optimization of the model set retrieval app is a major option for future work. Reducing overhead by making sure the most efficient data types or sorting algorithms are used, might make the retrieval process more user-friendly by taking up less time. As already mentioned, micro-optimizations in code were not the scope of this work and are therefore left for future development.

Another question left open by this work is the handling of the weather-related features when working with prediction horizons. Mentioned in an earlier chapter, the option to use data from a weather forecast was mentioned. As an example, let's assume a model that is to predict the parking occupancy in an hours' time (i.e. with a prediction horizon of 60 minutes) is using both temperature as well as humidity among other features. One option would now be to request forecasted weather data from an external source and use it as input for the model. If for example rain was forecasted, the model could use this increase in humidity and decrease in temperature to potentially make a more precise prediction about the parking occupancy. While in theory, this approach might seem valuable, there are various loose ends that might decrease the models' quality when using weather forecasts. For one, the forecast might of course be wrong and would therefore shift the models' prediction in the wrong direction. Secondly, reliable forecasts for small prediction horizons like 10 minutes might even be hard to come by or could cause a larger latency and data usage than just relying on current weather data. Especially for relatively small prediction horizons it might therefore make more sense to use current data on temperature and humidity than risking falsely shifted predictions by relying on forecasted data. However, this debate is a good starting point for any future work on this project. Comparing the performance and resource utilization of models using current weather data with models using weather forecasts might be a research question worth examining.

Regarding the design of the scores that were made up for this project, many decisions, assumptions, and restrictions had to be made. Different considerations led to following certain ideas and having to neglect others. One of those disregarded ideas was to come up with global scores that make it possible to compare a model with any other model from the entire database, across different requests. In the current implementation, each score (single model scores like the Resource Awareness Score, as well as scores of a model set like the Aggregated Model Score) must be seen in the context of the underlying request. For instance, the Performance Score of a model that is predicting the occupancy for parking lot 38 using a prediction horizon of 60 minutes, cannot be compared to a model predicting for parking lot 634 on a prediction horizon of 10 minutes. The reason for this is, that each score is normalized using the best possible result (i.e. in this case the best possible accuracy value for example) inside a specific request. A comparison across different requests, and therefore across different parking lots, prediction horizons, and window sizes might very well be valuable: It makes different requests and use cases comparable with each other, making it apparent where the quality of models might still be lacking. Global scores like this could be implemented by looking at the metrics of every model in the entire model database, assessing their scores, and using them as the basis for normalization. Nevertheless, it was decided against the implementation of global scores. By comparing every model with each other, the scale values would significantly be shifted. As a result, differences e.g. in the Performance Score between models would suddenly seem to be marginal, because of potential outliers. While comparing models inside a defined context would make differences in performance or resource awareness apparent, using global scores could make these differences vanish. Keeping the scale in a limited but meaningful way when comparing models of the same request is what makes

these comparisons valuable. Additionally, by returning the absolute value of the chosen performance metric gives the user at least some option for global comparison: While with the returned scores, a comparison across parking lots, prediction horizons, or window sizes is not possible, the returned absolute performance metric, e.g. the RMSE value is in fact suited for comparison in different contexts.

One last additional idea that could be developed is a change in the aggregation function of the introduced scores. As shown by Fagin et al., using the minimum value of an object's properties as its' overall score is a common practice [14]. In the context of this work, this could mean that the Model Score of a model would always be whatever single-model score - Resource Awareness Score or Performance Score - is smaller. This restructuring would not only make the use of weights obsolete but would also come with another wide array of implications, which would have to be examined beforehand. Ultimately, this work offers a wide range of points to improve upon and to take as an opportunity to do further research. However, the implemented training-pipeline and model set retrieval system, the designed metrics and scores as well as the conceptualized ideas brought up in this work can be considered as a valuable basis for the model selection and model ensemble process.

6.2 Conclusion

This work elaborated the theoretical background of model set creation and -retrieval and proposed both the design and implementation of a working model set retrieval system. In addition, a working model training pipeline was introduced, which acts as an important cornerstone in the subsequent creation of model sets. The showcased algorithms and ideas were then evaluated using different metrics such as the number of accesses, execution time, and network utilization.

The implemented preprocessing pipeline makes the overall model creation process much faster, by getting git of redundancies and making sure each preprocessing step is only done as often as it is needed. By already having preprocessed data available, the training of models can be done much more flexibly and less time-consuming. Utilizing the Weka library, the developed training pipeline then allows the user to have a large number of machine learning models trained and stored, according to their preferences in metrics. The structure of the pipeline makes it easy to both change the different values each model metric should assume (e.g. "models with window sizes of 1 and 5 minutes should be trained") as well as change the overall metrics that should show a variation (e.g. "apart from the window size, also the train-test-strategy for the models should change"). By combining the preprocessing- and the training-pipeline, the model creation process has been vastly automated and simplified. By setting up the desired metrics and their values and running the pipelines in the background, e.g. by using a virtual machine, a diverse model database with several thousand entries can be expected in only a few hours' time.

The main part of this work is the developed model set retrieval system. Using the models that were created with the help of the training pipeline, various model sets can be created and eventually be compared to one another. According to the users' settings, the best model sets are then selected and returned. The implementation allows the model set retrieval process to be largely customized to the users' liking. Not only is it possible to determine the prediction horizons or the window sizes of the demanded models, but the user is also able to select a preferred top-k algorithm and a performance metric that is used to select the best models and model sets. This work has implemented one naïve top-k algorithm along with two more sophisticated algorithms that make the selection of the best objects more efficient. The user is able to balance performance and resource awareness to their will, by

assigning different values to the two implemented weights. By determining an aggregation function for the QSLs inside a model set, the user is left with additional options regarding the assessment of resource awareness inside a model set.

Without having established a theoretical foundation, this implementation however would have been not possible. Researching different ideas on how to represent concepts in a correct but also easily understandable way probably took up the most amount of time and effort spent on this work. This project explored several proposals on how to measure and assess resource awareness, discussing their advantages and disadvantages and explaining why certain alternatives were chosen. By introducing the number of features a model uses as a form of intra-model resource awareness, the degree of efficiently utilizing limited resources can be assessed before forming any model sets. The distinction between intra- and inter-model resource awareness helps the model set creation process to be more precise and less redundant. In addition, the user can make more accurate observations about the resource utilization of a model set, by not only looking at the number of shared characteristics like window size or features within a set but also taking the number of used features into account.

Similarly, the concept of the implemented prediction horizon turned out to be a crucial part of the whole project. While in earlier stages of the implementation, both training pipeline as well as model set retrieval system were working as intended, the true predictive values of the models were questionable. Models that were only predicting the parking occupancy value for the current time slot, would be of little use for actual parking prediction. In addition, the implemented 24 hour-shift would only produce predictions for the next day, which turned out to be rather inaccurate as well as unpractical for actual usage. The idea of introducing a prediction horizon to the system because of these circumstances makes the parking availability prediction much more accurate and realistic. The created models are now able to predict parking availability in practically useful time intervals without having to compromise for prediction performance. Ultimately, these introduced concepts lead to a more sophisticated system that is focused on producing valuable results for real-life applications.

The developed system mainly acts as a form of multi-object optimization by weighting the performance of models or model sets against their resource performance. This aligns also with the main aim of this work: To balance performance against resource awareness in order to optimize the model set selection process. However, the implemented system can also be used as a constrained optimization problem by utilizing the setting `combineSameFeatures` in the API calls [16]. Instead of using multiple metrics (i.e. performance and resource awareness) for the model set selection, the constraint of only using model sets that have the same feature set could be set by activating `combineSameFeatures` and setting both weights to 1, indicating a 100% weight on the performance metric. Thereafter, the system would look for the best-performing model sets, under the condition of only considering model sets that share the same features. This shows the flexibility of the implemented system and how requests can be highly customized to the user's demands.

While the presented modules - both training-pipeline and model set retrieval system - were developed with a specific use case in mind - mainly the parking availability prediction use case - the conclusions derived from this study can and should be applied to other contexts in the field of machine learning model selection. Hence, the provided findings can also be applied to the cattle activity recognition use case to train models, create model sets, and eventually select the best model sets using a top-k algorithm.

Abbreviations

API	Application Programming Interface
CAR	Cattle Activity Recognition
CRUD	Create, Read, Update, Delete
FA	Fagin's Algorithm
HTTP	Hypertext Transfer Protocol
KNN	K-Nearest-Neighbors
MAE	Mean Absolute Error
MSE	Mean Squared Error
NA	Naïve Algorithm
QSL	Query Sharing Level
REST	Representational State Transfer
RMSE	Root Mean Squared Error
TA	Threshold Algorithm
VPN	Virtual Private Network

Figures

Literature

[1] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther, "Process mining: A two-step approach to balance between underfitting and overfitting," *Software & Systems Modeling*, vol. 9, no. 1, pp. 87–111, Jan. 2010.

[2] O. A. Montesinos López, A. Montesinos López, and J. Crossa, "Overfitting, Model Tuning, and Evaluation of Prediction Performance," in *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, O. A. Montesinos López, A. Montesinos López, and J. Crossa, Eds. Cham: Springer International Publishing, 2022, pp. 109–139.

[3] M. Rapp, R. Khalili, K. Pfeiffer, and J. Henkel, "DISTREAL: Distributed Resource-Aware Learning in Heterogeneous Systems," Apr. 2022.

[4] D. Preuveneers, I. Tsingenopoulos, and W. Joosen, "Resource Usage and Performance Trade-offs for Machine Learning Models in Smart Environments," *Sensors*, vol. 20, no. 4, p. 1176, Jan. 2020.

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[6] Z.-H. Zhou, *Ensemble Methods: Foundations and Algorithms*, ser. Machine Learning & Pattern Recognition Series. Boca Raton, Fla.: CRC Press, Taylor & Francis, 2012.

[7] T. G. Dietterich, "Ensemble Methods in Machine Learning," in *Multiple Classifier Systems*, G. Goos, J. Hartmanis, and J. van Leeuwen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, vol. 1857, pp. 1–15.

[8] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, no. 1, pp. 1–39, Feb. 2010.

[9] F. Eibe, M. A. Hall, and I. H. Witten, "The WEKA Workbench," in *Data Mining: Practical Machine Learning Tools and Techniques*, fourth edition ed. Morgan Kaufmann, 2016.

[10] W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and Jupyter*, 3rd ed. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly Media, Sep. 2022.

[11] M. Sünkel, G. Elmamooz, M. Grawunder, P. T. Hoang, E. Rauch, L. Schmeling, S. Thurner, and D. Nicklas, "Resource-Aware Classification via Model Management Enabled Data Stream Optimization," in *2022 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2022, pp. 23–33.

[12] A. Dehghani, O. Sarbishei, T. Glatard, and E. Shihab, "A Quantitative Comparison of Overlapping and Non-Overlapping Sliding Windows for Human Activity Recognition Using Inertial Sensors," *Sensors (Basel, Switzerland)*, vol. 19, no. 22, p. 5026, Nov. 2019.

[13] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 6th ed. New York: McGraw Hill Higher Education, Jan. 2010.

[14] R. Fagin, "Combining fuzzy information: An overview," *ACM SIGMOD Record*, vol. 31, no. 2, pp. 109–118, Jun. 2002.

[15] V. S. Marco, B. Taylor, Z. Wang, and Y. Elkhatib, “Optimizing Deep Learning Inference on Embedded Systems Through Adaptive Model Selection,” Nov. 2019.

[16] M. Feurer and F. Hutter, “Hyperparameter Optimization,” in *Automated Machine Learning: Methods, Systems, Challenges*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds. Cham: Springer International Publishing, 2019, pp. 3–33.

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Otto Friedrich Universität Bamberg festgelegt sind, befolgt habe.

Bamberg, den July 11, 2024

Niklas Diller