MASTER'S THESIS 2020

Usage pattern recognition for efficient pre-caching

Otto Sörnäs, Erik Gralén

ISSN 1650-2884
LU-CS-EX: 2020-25

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-25

Usage pattern recognition for efficient pre-caching

Otto Sörnäs, Erik Gralén

Usage pattern recognition for efficient pre-caching

Otto Sörnäs
ottosornas@gmail.com

Erik Gralén
egralen@gmail.com

June 22, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Rasmus Ros, rasmus.ros@cs.lth.se
Jens Argentzell, jens.argentzell@qlik.com

Examiner: Emelie Engström, emelie.engstrom@cs.lth.se

Abstract

Programs with many complex background calculations can have long loading times that frustrates users. One way to solve this problem is by performing all the necessary calculations before a user actually opens the program. This is called precaching, and is present in lots of different technology today.

This thesis presents a machine learning-based solution to a precaching-problem, applied in a cloud-based environment, by training two different sequential neural networks with labeled data. This approach relies on analyzing log files in order to concretize a usage pattern on which the neural networks can train. The networks were thereafter evaluated using a combination of MCC-score, confusion matrices, and an evaluation algorithm specifically written for this thesis.

Based on the results we can conclude that both of the machine learning models are potential solutions to this problem, albeit with their own strengths and weaknesses. The two networks differ in how computationally expensive they are to train, where the most expensive network also is the one that exhibits the best results. Nonetheless, the results for both networks are good enough to reduce the experienced loading time for a user.

Keywords: MSc, Machine learning, neural network, feedforward neural network, temporal convolutional network, TCN, FNN, prefetching, precaching, preloading, cloud

Acknowledgements

This master thesis was done at Qlik in Lund. At least until the corona pandemic hit Sweden, then it was mostly carried out in quarantine. Regardless, we would very much like to thank the team at Qlik for their friendly reception and help, when needed.

A special thanks to Jens Argentzell and Hampus von Post at Qlik, who not only provided us with technical knowledge and a never-ending support, but also had the mental energy to endure meetings with us twice a week for the entirety of the master thesis.

Lastly, a huge thanks to Rasmus Ros, our supervisor at LTH, for your invaluable feedback, support, and directives.

Thank you all!

Contents

1 **Introduction** 7
1.1 Problem Description 8
1.2 Related Work 9

2 **Theory** 13
2.1 Machine Learning 13
2.2 Time Series Forecasting 14
2.3 Artificial Neural Networks 14
2.3.1 Feedforward Neural Networks 15
2.3.2 Temporal Convolutional Networks 15
2.4 Model training 17
2.4.1 Autocorrelation 17
2.4.2 Hyperparameter tuning 18
2.4.3 Loss functions 18
2.4.4 Time Series Cross-Validation 19
2.5 Evaluation 20
2.5.1 Confusion Matrix 20
2.5.2 Matthews correlation coefficient 21

3 **Approach** 23
3.1 Research Method 23
3.2 Problem identification and motivation 24
3.2.1 Model requirements 24
3.2.2 Data Requirements 25
3.3 Implementation Method 25
3.4 Data collection and analysis 26
3.4.1 Data Preparation 27
3.4.2 Feature Engineering 27
3.5 Evaluation and comparison 29
3.5.1 Data simulation 30

3.5.2 Model evaluation 30
3.5.3 Comparison 31

4 Implementation 33
4.1 Feedforward Neural Network 33
4.1.1 Implementation 33
4.1.2 Results and evaluation 35
4.2 Temporal Convolutional Network 39
4.2.1 Implementation 40
4.2.2 Results and evaluation 41
4.3 Static Rules 42
4.3.1 Implementation 44
4.3.2 Results and evaluation 45

5 Results 49
5.1 Evaluation 49
5.1.1 FNN 49
5.1.2 TCN 50
5.1.3 Static rules 50
5.2 Comparison 50

6 Discussion 53
6.1 Results discussion 53
6.2 Limitations 54
6.3 Internal and external validity 55

7 Conclusion 57

References 59

Chapter 1
Introduction

Precaching (also called prefetching or preloading) is the notion of fetching information before the processor requests it. This technology works by analyzing usage patterns of the users, and make sure applications predicted to soon be launched already have loaded necessary files to memory beforehand. This reduces loading times and consequently creates a more pleasant user experience. The software environment subject to this master thesis is the Qlik Sense environment. This is a web based tool for data visualisation and data analytics. Qlik Sense indexes relationships in data in a way that allows the user to gain complex insights in the data. An instance of Qlik Sense has several user-created applications, and opening an application requires the server to load all of the data into its RAM and subsequently perform calculations. For applications with large amounts of data, this step can take a very long time. However, once the application is loaded, any subsequent users do not have to reload it.

Analyzing the usage pattern of a single user in order to predict which application is thought to be opened when is a quite established area of expertise. This is actually a technology widely used in today's everyday life, even though we might not be particularly aware of it. Most notably it is used in all Windows and Linux machines, through the SysMain (previously knows as SuperFetch) [29] and preload [12] programs respectively. However, more and more applications are becoming web based with lots of users. To predict application launches in a cloud environment with multiple users is not quite as an explored area, and this master thesis aims to present an easy to implement but yet effective solution to this problem via machine learning.

In order to analyze the usage pattern of the Qlik Sense applications, we have used time series analysis and time series forecasting. Time series analysis can be described as a way of trying to understand trends in time series data, and time series forecasting can be described as making predictions based on the analysis. Two different machine learning models, namely a Feedforward Neural Network (FNN) and a Temporal Convolutional Network (TCN), will be implemented in order to carry out the forecasting, and the results will be compared to each other.

The results show that both of these networks are satisfactory solutions to the aforemen-

tioned problem. FNN is a more lightweight network which is easy to train, albeit it does not have an outstanding prediction accuracy. TCN, on the other hand, has a higher potential accuracy, while being harder and more computationally expensive to train. It is therefore recommended to use FNN where memory constraints are of importance, and TCN when a high accuracy is preferred.

The report is outlined as follows. Chapter 2 introduces the theoretical background necessary to understand the work done in this thesis. Chapter 3 describes how the theoretical parts are applied practically, and through which methods. Furthermore, Chapter 4 describes how the implementation for the different models were carried out, and thereafter presents some preliminary results. In Chapter 5, the different results summarized, and then compared to each other. Chapter 6 discusses limitations, internal and external validity, and the implementation phase. Lastly, final thoughts, a summary of the findings, and suggested future work is presented in Chapter 7.

1.1 Problem Description

The major scope of this master thesis is therefore to research, analyze and review how machine learning algorithms can be applied in creating an adaptive system that can predict which applications should be precached, and when. More concretely, the goal is to create a system which can be applied to any given Qlik Sense instance. By doing this we will also analyze and get an understanding for how usage patterns of applications are opened and used in a production environment. In order to reach these goals, the following will be answered:

- How can a machine learning model be used to predict user behavior for precaching cloud applications?
- How can the performance of the predictive model be evaluated?
- What is a sufficient degree of prediction accuracy to have a real world impact?
- How does the predictive model perform compared to a manually created ruleset?

The Qlik Sense platform is, as previously mentioned, a web based business intelligence tool which primary goal is to allow a user to gain complex insights in data. This is done through user created apps, which can contain quite intricate calculations. When a user opens an application, thus starting a new session, the data is loaded into the server's memory and the calculations for that specific app are performed. Most of this information is subsequently stored in memory for eight hours after session ends. Eight hours is a default settings value, but it can be changed via the administrator interface. There exists a cloud based version of Qlik Sense where these settings works differently, however this is not the Qlik Sense version subject to this master thesis. When another user opens the same app during this eight hour long time frame, the loading time for that app opening is drastically reduced since all the necessary information is already stored in memory. This app opening initiates a new session, and the app is instead stored in memory for eight hours after this new sessions ends.

Qlik offers the possibility to schedule apps for precaching [27], meaning an app can be loaded preemptively before the first user of the day accesses it. Currently, this scheduling is done manually by the users themselves, but by applying an adaptive learning algorithm to the

usage pattern of the analytics platform it could be optimized and automated to understand which applications should be opened and when.

Whenever a new session is started, information is stored as rows in a session log file. Some of the information stored is the timestamp for when a new sessions was initiated, the length of the session, which user initiated the session, and which app was opened. A new log file is created every day, which contains all the session information for the given instance of Qlik Sense.

These session log files are what is going to be used to generate input data for the machine learning models in this master thesis. How these logs are parsed and prepared are further explained in Section 3.4.1. The purpose of the machine learning models is to generate a list of timestamps where it is explained which apps to precache when. This list of timestamps is thereafter to be used as input for a test service, provided by Qlik. This test service is able to act as a scheduler for precaching apps if given the correct input parameters.

Figure 1.1 shows how this cycle works under the hood. A request to open a specific app is sent to the Qlik Indexing Engine (QIX) Engine. This engine subsequently loads the correct data, and performs the necessary calculations for that app. Information regarding the current session is stored in the log storage. The precaching feature developed in this thesis, named "Pattern Recognition" in the figure, accepts these logs as input. The output, in form of a list of timestamps, is thereafter sent to the test service. This test service thereafter schedules the app openings accordingly.

1.2 Related Work

There has been previous work done within the area of event prediction via time series analysis. Callara and Wira [5] present a probabilistic approach to predict when a user will launch an application in a cloud environment, and consequently reduce the launching time. This is however a presentation of a statistical model, and it does not make use of any existing machine learning models. The theory behind it is something that might particularly useful though, since they describe how to estimate Probability Density Functions (PDF) and use a Kernel Density Estimation (KDE) based on periodic patterns of the user's activity.

The company Uber, for example, is making use of probabilistic time series forecasting to predict the number of trips during special events, such as big sports event. Zhu et al. [39] present a Bayesian model of this, and apply it to large-scale time series anomaly detection at Uber. Laptev et al. [18], also working for Uber, present another solution using long short-term memory (LSTM) in order to create an accurate time-series forecast during high variance segments.

In both Linux and Windows, as previously mentioned, a prediction-based prefetching scheme is used to store applications in the RAM that are thought to soon be launched by the user. The program used for Linux is called `preload` [12], and is using a first-order Markov prediction model to learn the usage pattern. Song et al. [34] present a usage pattern-based scheme for prefetching application launches on mobile devices. To predict the next application launch, Song et al. [34] proposes a Window and Weighted Sum-based (WWS) prediction scheme.

All of these papers give examples for how event prediction can be executed and implemented with the help of machine learning, and will be relevant in one way or the other. Either

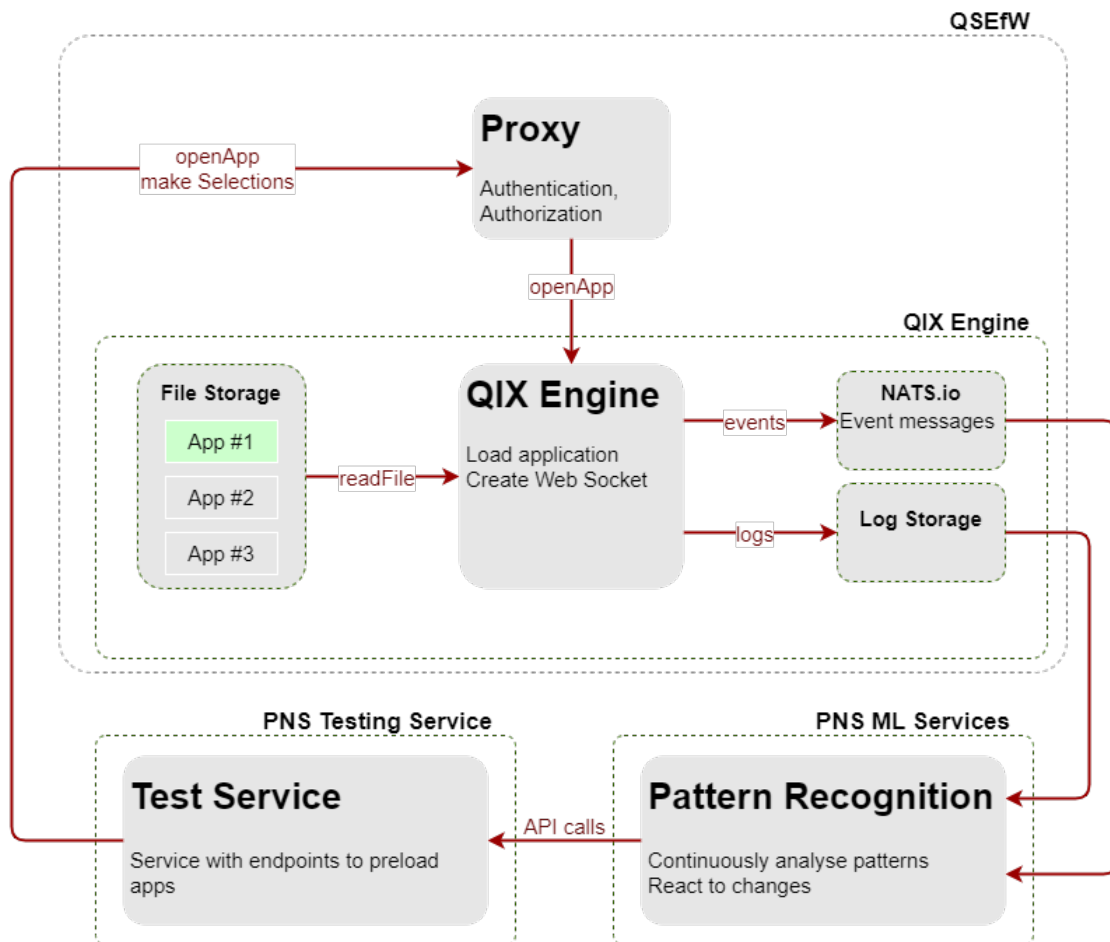


Figure 1.1: An overall view of the platform architecture. The pre-caching feature developed in this thesis is what is called "Pattern Recognition" in the figure. The output is then fed to the test service, which schedules the app openings.

as a starting point for future research, or by actually providing something that can be directly included in this master thesis. However, Liao et al. [19] present a paper wherein they develop several machine learning models in order to precache app openings in a data centre. They compare their machine learning models to already established rules, and thereafter evaluate their findings. This is very close to what we are trying to achieve in this thesis. Liao et al. [19] concludes that their machine learning models outperforms the rules already in place, and that machine learning based precaching schemes are a good way to optimize performance.

Chapter 2

Theory

In order to give the reader a basic understanding of the methodology used in this master thesis, the theory behind the methods will be presented in this chapter. First, machine learning and time series forecasting as research fields are introduced. The two different machine learning models used in this master thesis project are thereafter introduced and explained. Lastly, the different techniques and methods used to tune the models' parameters and then evaluate the results are explained. These methods are subsequently used in Chapter 4, and are therein thoroughly explained from a practical point of view.

2.1 Machine Learning

Machine learning originated over 50 years ago as a subdiscipline to the Computer Science and Statistics fields. In present times the methodology is mainly used to analyze data in order to uncover data patterns. These patterns then serves as a basis for decisions, or are used in order to predict future data [22]. Machine learning is used today in very varied technologies, for example speech recognition, stock market predictions and computer vision. There are mainly three different types of machine learning in use: reinforcement learning, supervised learning and unsupervised learning. In this project, however, only supervised learning will be used.

Supervised learning, or predictive learning, tasks the agent with learning the mapping from x to y, given a training set made of pairs (x_i, y_i). Usually, a part of the training set is set aside in order to evaluate the predictive performance on test examples. In other words, the output data is known from the beginning, and the agent is used to identify the connections between input (x) and output (y) [6]. Supervised learning is mainly used for either classification of data, or for regression. The problem in this thesis is a classification problem, which is why supervised learning is used.

2.2 Time Series Forecasting

A time series is a time-oriented or chronological sequence of observations of a variable [23], where the variable typically is collected at equally spaced time periods. A classic example of a time series would be a diagram showing how the stock market prices changes over time. This is also a scenario where one might want to forecast the time series, i.e predict how the future stock market development will look like.

In order to create an accurate prediction of how future time series development will look like, a time series analysis need to be conducted. The data is broken down into smaller constituents, and an effort is made in order to spot visible underlying patterns and data trends. This is a large and vital part of the forecasting process, since it is here the important features and data relationships are identified [23]. How this was conducted in this master thesis project is more thoroughly described in Section 3.4.

Quantitative forecasting techniques uses historical time series data together with a forecasting model. This model then uses the data together with the previously identified patterns and data relationships to find statistical relationships between previous and current values of the variable. These patterns and relationships are subsequently used to extrapolate past and current behaviour into the future [23]. Machine learning models can advantageously be used for this purpose.

2.3 Artificial Neural Networks

Machine learning as a whole is made up of large different families of different machine learning models. One of the most common model to use when it comes to classification problems is the artificial neural network (ANN). The name is derived from the fact that the ANN is vaguely inspired by the biological neural networks that exists in brains [20]. An ANN consists of, as the name implies, a network of artificial neurons. These so called neurons are nothing more but computational nodes, which collectively learn from the input in order to optimise its final output [24]. How a single artificial neuron is constructed is shown in Figure 2.1. As can be seen in this figure, a neuron consists of three main components: weights, a bias, and an activation function. These three components can be described as follows:

- *Weights.* An artificial neuron receives a set of inputs (x_1, x_2, x_3) and multiplies each input with a corresponding weight (w_1, w_2, w_3). The weight can be explained as showing the strength of a particular node, that is to say how much that node actually affects the output value.

- *Bias.* Biases in neural networks are extra neurons added to each layer, which store the value of 1. These bias neurons also have weights attached to them. This makes it possible to move or "translate" the activation function left or right on the graph.

- *Activation function.* The weighted inputs and a bias b is then summed and an activation function is applied to the sum. The purpose of the activation function is to convert an input signal of a node to an output signal. The activation function can be chosen to be any function such as a linear, binary, or sigmoid activation function, depending on what format you want the output data to be in.

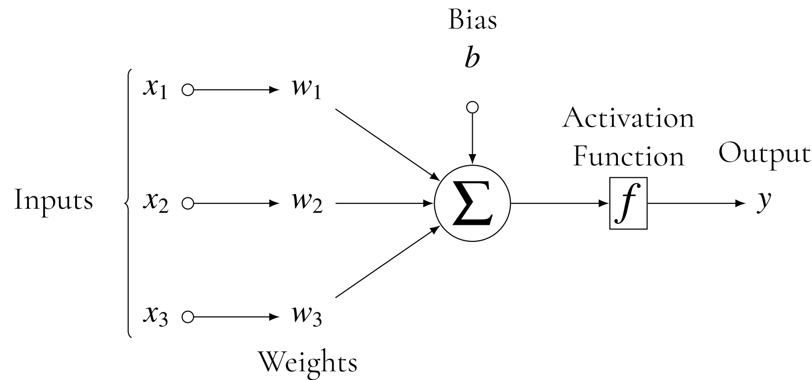


Figure 2.1: An overview for an Artificial Neuron. The input data, x, is multiplied with a respective weight, w. Everything is summed up together with a bias, and then an activation function is applied to data in order to create an output, denoted y.

An artificial neuron's output can then be used as input to one or more other artificial neurons. These neurons together make up a neural network, and make up the basis for many different machine learning models.

2.3.1 Feedforward Neural Networks

Depending on how the neurons in an ANN are structured, one can design different neural networks for different purposes. One of the most common, and perhaps also one of the most simple ways of organizing these artificial neurons is the feedforward neural network (FNN). This network consists of several sequential layers, which in turn consists of several artificial neurons. As shown in Figure 2.2, an FNN consists of at least three layers: an input layer, an output layer, and any number of hidden layers. Having multiple hidden layers stacked upon each-other is commonly called deep learning [24]. The reason why this network is one of two chosen networks for this thesis is due to fact that it is an established way of solving this type of classification problems. It should therefore be able to function as a type of baseline.

Another important point to note here is that each of the hidden layers can have a different activation function. Choice of the activation function to be used depends on the problem in question and the type of data being used. For a neural network to make accurate predictions, each of the neurons needs its weights fine tuned at every layer. Tweaking the weights in order to achieve more accurate results is what is more commonly known as the "learning" part in machine learning. The algorithm through which they tune the weights is called back propagation, and is further covered in Section 2.4.3.

In an FNN, each neuron's input is the output from every neuron in the previous layer, except for neurons in the input layer which receive input variables directly.

2.3.2 Temporal Convolutional Networks

Up until quite recently, deep learning practitioners have used recurrent neural networks (RNN) as the default approach for sequence modeling (or time series forecasting) [1]. The rea-

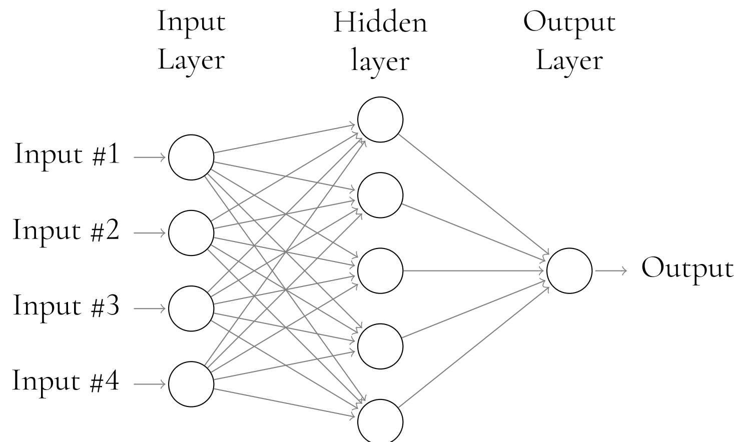


Figure 2.2: An overview over a simple simple Feedforward Neural Network. Each layer consists of several artificial neurons. Each neuron transmits its output to the next layers. After the signal has traversed all layers, an output is generated.

son why RNNs are considered good at especially sequence modeling is partly due to the fact that they can take inputs of variable length, and also because they have cells which functions as a memory [9]; they can consequently remember and learn from previous inputs because of this [21]. Examples of popular RNNs for sequence modeling would be the Long Short-Term Memory (LSTM) [31] and Gated Recurring Units (GRU) [9]. However, in 2016, Google DeepMind published a paper called "Pixel Recurrent Neural Network" [36] wherein the authors showed that it is instead possible to use a Convolutional Neural Network (CNN) as a sequence model (instead of an RNN), while still achieving high prediction accuracy.

CNNs can be described as an extension of the Feed-Forward Neural Network, where some of the hidden layers are convolutional layers [10]. Generally speaking, CNNs are hierarchical and RNNs sequential architectures [38], and this is one of the major reasons as to why anyone would want to use a CNN as a sequence model instead of an RNN. Because unlike in RNNs where the predictions for later timesteps must wait for their predecessors to complete, convolutions can be done in parallel since the same filter is used in each layer [1]. Networks using a CNN as sequence model are more commonly known as Temporal Convolutional Networks (TCN). Facebook also released their own TCN in 2017, called Fairseq [25], and claims it runs nine times faster than their RNN benchmark.

Bai et al. [1] conducted an empirical evaluation of TCNs in 2018, and the authors note that TCN have two distinguishing characteristics:

1. The convolutions in the architecture are causal, meaning that there is no information “leakage” from future to past.

2. The architecture can take a sequence of any length and map it to an output sequence of the same length, just as with an RNN.

Their evaluation consists of a series of benchmark competitions of TCNs versus RNNs, LSTMs and GRUs. This is done on eleven tasks that have been commonly used to benchmark the performance of different RNNs. Their result is that TCNs are not only faster, but

they also produce greater accuracy in nine cases (and tied in one) [1]. They end the evaluation by presenting the reader with a list of advantages of TCNs [1]:

- Since convolutions can be done in parallel in TCN, a long input sequence can be processed as a whole in TCN, instead of sequentially as in RNN. This speeds up the training process quite a bit by shortening both the training and evaluation cycles.
- TCNs offer more flexibility in changing its receptive field size, principally by stacking more convolutional layers, using larger dilation factors, or increasing filter size. This offers better control of the model's memory size.
- TCNs have stable gradients, which means they avoid the problem of exploding/vanishing gradients, which is a major issue for RNNs. Since the gradients control how much the network learns during training, if the gradients are very small or zero, then little to no training can take place, leading to poor predictive performance. This is called the vanishing gradient problem. The opposite is true for exploding gradients, when large error gradients accumulate and result in very large updates to neural network model weights during training.
- Compared to RNNs, especially in the case of a long input sequence, TCNs have a very low memory requirement for training.

According to the authors [1], there is one notable disadvantage to using TCNs that might have an impact on this master thesis: TCNs might require more memory during evaluation. This will be further explored and examined in Section 4.2.1.

2.4 Model training

Training a machine learning model consists of several steps. Firstly, it is required to identify if the data is usable at all. If it is indeed usable, it is thereafter possible to identify the hyperparameters that yield the best result with respect to the task and setup. Lastly, it is time to actually train the machine learning model. The theory behind these different steps are explained in this section.

2.4.1 Autocorrelation

In order to train a model on a dataset, you need to first figure out if there exists a pattern in the data (that the data is non-random), and where this pattern exhibits its strongest correlations. If the data is close to being random, it is neigh impossible to predict future data sequences. To figure this out, something called the autocorrelation function can be used. The autocorrelation function [3] is mainly used for two reasons:

1. To detect non-randomness in data.
2. To identify an appropriate time series model if the data are not random.

If there exists a set of observations $Y_1, Y_2, ..., Y_N$ at times $X_1, X_2, ..., X_N$ (where the time between the observations are assumed to be equal) the autocorrelation can be calculated as follows:

$$r_k = \frac{\sum_{i=1}^{N-k}(Y_i - \overline{Y})(Y_{i+k} - \overline{Y})}{\sum_{i=1}^{N}(Y_i - \overline{Y})^2} \tag{2.1}$$

The autocorrelation is not the correlation between two variables, it is the correlation between two values of the same variable at times X_i and X_{i+k} [3]. The value of k is the time gap being considered, and it is called the *lag*. A lag 1 autocorrelation, i.e when $k = 1$, is the correlation between values that are one time period apart. So generally speaking, the lag k is the correlation between values that are k time periods apart. When this is used to identify an appropriate time series model, the autocorrelations are usually plotted for many lags. If the observations are random, the autocorrelations should be near zero for practically all time-lag separations. If non-random, one or more of the autocorrelations will be significantly non-zero. The higher the value of a lag, the higher the correlation between the time periods. This can advantageously be used to identify which lags to use when training a machine learning model to forecast a time series.

Uncorrelated, however, does not necessarily mean the data is random. Data that does not show significant autocorrelation can still exhibit non-randomness in other ways; auto-correlation is just one way of measuring randomness. Though, checking for autocorrelation is typically a sufficient test of randomness when validating a model.

2.4.2 Hyperparameter tuning

For neural networks, there are many parameters that are adjustable which can affect on the resulting performance of the network. Some of the parameters that can be adjusted include number of layers, number of neurons in each layer, the activation functions, learning rate, and number of epochs. Considering all of the hyperparameters, the number of possible combinations is incredibly large. Since there exist no closed-form solution to choose hyperparameters, the only approach is to use a trial-and-error approach [33]. Luckily enough, there exists open source libraries that automates this process and finds the optimal combinations of parameters for you. The parameter optimization used in this master thesis was the Bayesian optimization from Scikit-Optimize [15], a sequential model-based optimization built on Scikit-Learn.

The results from this optimization algorithm was subsequently used for choosing the parameters when implementing the models, as described in Section 4.1.1 and Section 4.2.1 respectively.

2.4.3 Loss functions

In neural networks, the individual weights and biases are estimated through training, meaning that they can be adjusted so that the network provides accurate answers [23]. To evaluate the performance of a neural network, a loss function is typically used. Broadly speaking, loss functions can be classified into two major categories depending on the type of learning task we are dealing with: regression- and classification losses. In classification, we are trying to predict an output based on a set of finite categorical values. Regression, on the other hand,

deals with predicting a continuous value - for example forecasting a numerical quantity. The goal of training a network is to adjust the trainable parameters such that the loss function is minimized.

Often a procedure known as backpropagation is used together with a loss function to estimate the neural network's parameters [23]. When a feed-forward neural network accepts an input x and produces an output y, information flows forwards through the network and produces a cost, generated from the loss function [2]. This is called forward propagation, and the backpropagation algorithm allows the cost to flow backwards through the network in order to compute the gradient of the loss function [13]. Backpropagation refers only to the method for computing the gradient, while another algorithm (such as the stochastic gradient descent [2]) is used to adjust the trainable parameters using the gradient. The function used to adjust the weights is called an optimization algorithm [28].

2.4.4 Time Series Cross-Validation

Historically, the data used for training a machine learning model has been split into three parts: a training, test, and validation set. The model learns from the training data set and is then evaluated against the validation data set. The model's parameters are subsequently adjusted according to how well the model performed on the validation set, and the model in its entirety is finally evaluated against the test data to see how well the final model is performing. There are other ways of calculating an unbiased estimate of model skill on unseen data. One popular example is to use cross-validation to tune model hyperparameters instead of a separate validation dataset.

Cross-validation is a model validation technique for asserting how well a machine learning model will adapt to independent data. This is used in order to understand how accurate the model's predictions will be in practice. A very common approach to cross-validating predictive machine learning models is divide the data into k number of chunks. Of the k number of chunks (or folds), one is separately saved as a validation set of data, and the remaining $k - 1$ number of folds are used to train the model. The training is then repeated k number of times, where every fold is used as a validation set once. This approach is called k-Fold Cross-Validation. The k number of results can then be averaged which equals to the final result, revealing how the model is performing.

The issue with k-Fold Cross-Validation is that you have to assume that there is no relationship between the observations, i.e. each observation is independent. This is not the case with time series data, where the temporal order in which values were observed is important. To remedy this, a procedure sometimes known as "evaluation on a rolling forecasting origin" [16], or "rolling cross-validation", can be used. The data is once again divided into k number of chunks, where the first chunks are used to forecast later data points. The same forecasted data points are then included as part of the next training dataset and subsequent data points are forecasted. The average accuracies of the test folds are then computed in order to evaluate the model's performance. This could be viewed as a k-Fold Cross-Validation where the training happens for k *consecutive* time folds. The Figure 2.3 illustrates how the rolling cross-validation works.

The result from each iteration of the cross-validation can then be used to tune the parameters of the model in order to boost its performance.

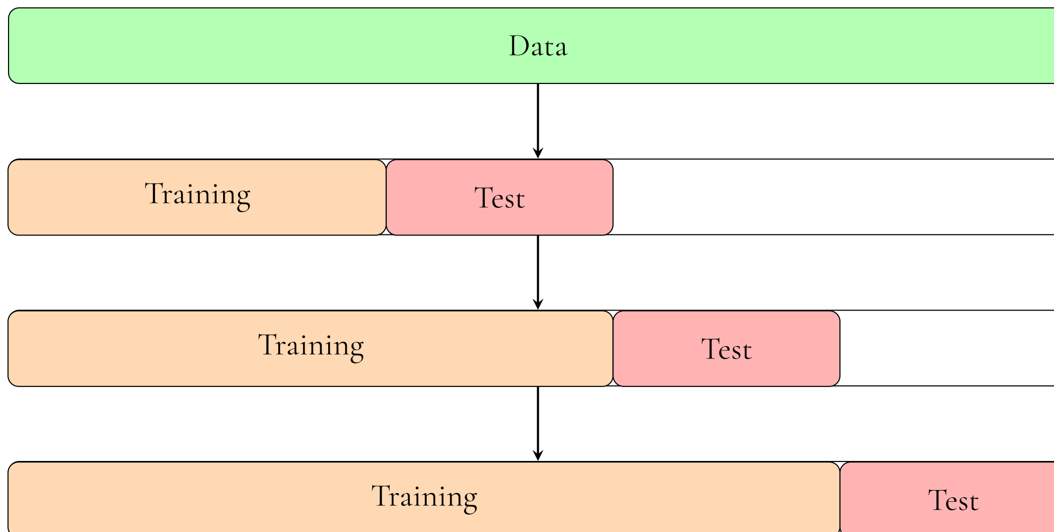


Figure 2.3: An overview for how time series cross-validation works. The original data set is divided into a training and test set. The test set is then included in the new training set, and this continues until all data is traversed.

2.5 Evaluation

An important finalizing step of the machine learning process is to evaluate the results gained in a sufficient manner. Two different evaluation techniques were utilized during this master thesis; the confusion matrix, which is used as a visualization of the result, and the Matthews correlation coefficient, which generates a numerical values in order to approximate the model's prediction accuracy. In this section, the theory behind these two methods used for evaluation are explained.

2.5.1 Confusion Matrix

The problem in this master thesis was approached as being a binary classification problem. This means that the prediction done by the machine learning model only can take one of two values. It either predicts that a user is about to open the given application during the upcoming hour, or it predicts that a user will *not* open the application. This results in four different outcomes:

- Actual positives that are correctly predicted positives are called true positives (TP). I.e. when the model correctly predicts an application is about to be opened.
- Actual positives that are wrongly predicted negatives are called false negatives (FN). I.e. when the model misses to predict when a user opens an app.
- Actual negatives that are correctly predicted negatives are called true negatives (TN). I.e. when the model correctly predicts that no user is going to open the app in the upcoming hour.

- Actual negatives that are wrongly predicted positives are called false positives (FP). I.e. when the model wrongfully predicts a user to open the app.

Prediction outcome

		p	**n**	**total**
actual value	**p′**	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

Figure 2.4: An example of a generalized confusion matrix for binary classification. The top left and bottom right quadrants are the wanted results, i.e. when the model predicts correctly. The top right and bottom left quadrant represents when the model fails its classification.

All the outcomes can subsequently be summarized in what is called a confusion matrix [8], and a generalized confusion matrix is shown in Figure 2.4. This can advantageously be used to gain a visual overview regarding how well the model is performing.

2.5.2 Matthews correlation coefficient

More often than not, it is desirable to calculate a numerical value based on the confusion matrix in order to evaluate how well the model actually performed. Many researchers consider computing the accuracy as a reasonable performance metric [8]. Accuracy represents the ratio between the correctly predicted instances and all the instances in the dataset, and can be calculated as follows:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.2}$$

This is indeed a good estimation of the model performance, assuming there is a balanced dataset. However, when the dataset is unbalanced, i.e. a dataset in which one class is over-represented with respect to the others, accuracy cannot be considered a reliable measure anymore [8]. This is due to the calculation providing an overoptimistic estimation of the classifier ability on the majority class.

As described in section 3.4.1, the data used in this master thesis had to be upsampled in order to create a continuous time series. This means that a vast majority of the data points in the dataset are zeroes, i.e. signaling that the app is not open. An effective solution overcoming the class imbalance issue comes from the Matthews correlation coefficient (MCC), a special

case of the ϕ phi coefficient [8]. MCC takes into account the ratio of the confusion matrix size, and can correctly inform whether or not the prediction evaluation is going well or not [7]. MCC can be calculated the following way:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{2.3}$$

This function returns a number between -1 and 1, where a 1 is a perfect classifier, and -1 is a perfect *inverted* classifier, meaning the model got every single prediction wrong. A 0 represents random guesses, which means a good classifier would preferably score in the close proximity to a positive 1.

However, if one of the rows (or columns) in the confusion matrix (in Figure 2.4) consists of zeroes, the MCC would be undefined. For example, if the obtained values from the confusion matrix would be $TP = 37, FP = 18; TN = 0, FN = 0$, the denominator in Equation 2.3 would be 0. This might be a problem if the dataset is small *and* unbalanced.

Chapter 3
Approach

This chapter aims to explain the practical applications of the theory described previously in Chapter 2. Firstly, the general work process and research method are explained. The actions taken in every step of the research method are then explained, and linked together with the subsequent sections where these actions are described more in-depth. The general approach when implementing the different models is presented; from data collection and analysis to feature engineering. Lastly, the classification of the output, the model evaluation, and the model comparison are explained.

3.1 Research Method

This master thesis project aims to use design science as a general research method. Design science research (DSR) is an approach to performing research which aims to develop general design knowledge in a specific field [11]. Iivari et al. [17] notes that there is no widely accepted definition of DSR. However, their take on it, which we accept, is that DSR can be described as [17]:

> "a research activity that invents or builds new, innovative artifacts for solving problems or achieving improvements, i.e. DSR creates new means for achieving some general (unsituated) goal, as its major research contributions."

Iivari et al. also concludes that DSR is more of a research paradigm, within which one can use different research methods [17]. More concretely, Peffers et al. [26] divides DSR into six distinct activities, which should in some way be included in a DSR process. These six steps correspond to how the thesis has been carried out:

1. *Problem identification and motivation:* Define the research problem and justifying the value of a solution. This is described in Section 3.2.

2. *Define the objectives for a solution:* The objectives can be either quantitative or qualitative, however the objectives should be derived from the problem specification. The objectives for this master thesis were derived in conjunction with Qlik. The results from this is described in Section 1.1.

3. *Design and development:* How the design and development of the two machine learning models underwent is described in Section 3.3.

4. *Demonstration:* For each of the two machine learning models, preliminary results and usage of the implementations are described. This is done in Chapter 4.

5. *Evaluation:* The evaluation and comparison process is described in Section 3.5. The results from the subsequent evaluations are presented during the implementation description for each of the models in Chapter 4. The evaluations are thereafter presented as a whole and compared to each other in Chapter 5.

6. *Communication:* Communicate the usefulness and effectiveness of the artifact to other researchers and practicing professionals.

While Peffers et al. [26] notes that there is no need for a researcher to go through these steps in sequential order (one can instead start at almost any step and move outwards), we have in this thesis opted to do most of the steps in order. This thesis began with defining the objectives, and formulating a problem description. The data used in this thesis was thereafter processed, and the models implemented. Lastly, the models were evaluated. Based on the evaluation results, the models were re-implemented in order to tune the parameters and achieve an even better result. This iterative process is described more in-depth in Section 3.3.

3.2 Problem identification and motivation

This section corresponds to the first step of the design science process, as described in Section 3.1. First the model requirements will be described. This is basically the end goal; what we want our models to achieve. Thereafter, data requirements will be described. This is a brief description of the data used in this thesis, and the information it contains.

3.2.1 Model requirements

Since a pre-loaded application takes up memory space, it is of utmost importance to minimize the time an application is pre-loaded before being used the first time, all while not being pre-loaded too late or without being used at all. The implemented model therefore has to first and foremost identify *if* there exists a definable usage pattern for the given application, and thereafter concretize this pattern in order to be able to determine when the application should be pre-loaded. Trying to pre-load an application while it is already stored in memory does little to nothing besides bumping the CPU-usage a bit, which means opening an application already stored in memory is a non-wanted behaviour. The model consequently has to also be able to identify when an application is already opened, and opt to not pre-load it if so.

To realize these requirements, the model could be designed from two different approaches, each with their pros and cons:

- View the pre-loading as being an extension to an application, where the user could decide to activate and deactivate the feature at will. Given that there exists enough data for the given app, and that there actually exists a usage pattern, the model could be quite simple while still generating accurate results. Granted, having an individually trained machine learning model for every application might not be feasible storage wise.

- Implement a generalized machine learning model that analyzes all applications at once and produces a single homogeneous usage pattern. This would solve the possible storage problem, but the model would be much harder to develop, if doable at all. It would require a very thorough data analysis and a model that would be quite complex.

3.2.2 Data Requirements

Machine learning is, as previously stated, based on analyzing data in order to find patterns within it. This means that there is usually a need for a quite large amount of data to analyze, or in other words, a lot of data to train on. If this training set of data is not sufficiently large, there is an elevated risk of the model *overfitting*. Overfitting is when the model learns from the noise of the deviating data instead of the data signal itself [14]. Broadly speaking, the machine learning model becomes more accurate the more data it can train on.

The dataset used in this master thesis is provided by Qlik, and comes from their R&D department's production environment. It is provided in the form of system and session logs, and contain information about every new session a user starts. For example, a selection of information every row in the log files contains would be a timestamp for the session length, the session start time, and the name and ID of the application opened by the user. The logs used for this master thesis spans from the middle of October to present, resulting in about six months worth of data.

However, not all data in the log files could be used. The relevant data had to be parsed from the log files, and then restructured into a format that could be used for the purpose of training machine learning models. This meant that a lot of the time at the start of this thesis was allocated to creating a functioning data management infrastructure.

3.3 Implementation Method

The forecasting process by Montogomey et al. [23], was used when implementing the machine learning models and predicting application launches. This is a generalized model for machine learning implementation with a heavy focus on data analysis. The first five steps of the seven step long process can be seen in Figure 3.1. The last two steps were cut out since they involve launching and validating the model in a production environment, which we will not be doing in this master thesis project.

As previously mentioned in Section 3.1, DSR can be viewed as a research paradigm, wherein you can use other research methods. The forecasting process described in this section is one of these methods, and has in its entirety been used during the design and development

step of DSR in this master thesis. This means that while some of the steps in DSR and the forecasting process might display some similarities, or even share the same name, they refer to different things. The following text therefore aims to explain the steps in the forecasting process more in-depth, and to investigate the differentiations from DSR.

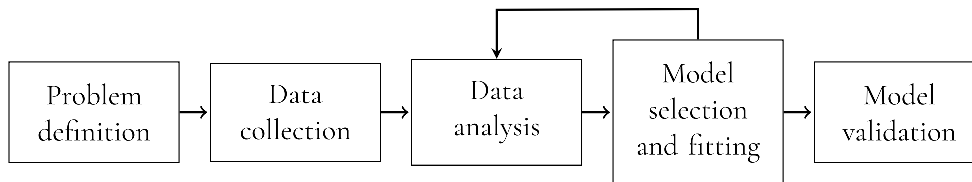


Figure 3.1: The parts of the forecasting process, as described by Montgomery et al. [23], which is going to be used in this thesis.

The problem definition step of the forecasting process refers to a more specialized approached than in DSR. This step more or less intends to create a framework on which you can base the machine learning model implementation. Questions to be answered are among others:

- What problem is this model trying to solve?
- What are the input and output parameters of the model going to be?

This was conducted the same way for both machine learning models used in this master thesis, and is explained in the introductory part of each model in Chapter 4. The data was also collected the same way for both models, and the way this was done is described in Section 3.4.1.

Data analysis, feature extraction, and model fitting is a cyclic process where each iteration of a trained model is evaluated. The first steps, data analysis and feature extraction, are explained in Section 3.4.2. The evaluation is done after the model validation, and constitutes the last step in the forecasting process. Time series cross-validation was used for for the validation, and is described in Section 2.4.4. The results from the cross-validation was subsequently evaluated, and the techniques used for the evaluation are described in greater detail in Section 3.5.

Based on this model validation, the features used to train the model might be changed. Identifying which data to use as features when training a machine learning model is a large and time consuming part of the process.

3.4 Data collection and analysis

Data collection and analysis is a very large, time consuming, and important part of the process when implementing a machine learning model. It is preparatory work which later acts as basis for model training. This part of the process is a cornerstone of the Design and development step in DSR, as described in Section 3.1.

3.4.1 Data Preparation

The data used in this project originates from a Qlik Sense instance used internally by Qlik. This instance generates several types of log-files where the most relevant log-files were the session logs. These logs provided timestamps on when users connected to specific applications and some general information about the session. When analysing the logs, it was found that there existed several bot accounts performing scheduled activities. It was consequently decided to remove all log entries belonging to these accounts in order to only train the model on human activities.

Since this master thesis aims to analyze the usage patterns of apps, the apps without a sufficient level of usage (or where the usage did not exhibit a clear pattern) were consequently filtered out. This was done by counting the number of log entries for each app, combined with analyzing the autocorrelations to ensure the usage pattern was non-random.

It is very important, but also very hard, to identify relevant history for the variables that are to be forecast. Often information collection and storage changes over time, and not all historical data are useful for the current problem. It is not rare to also encounter missing values of some values, or other data-related problems that may have occurred in the past. One example of this would be the fact that log entries only was created when a user started a new session, meaning that there was no data for whenever a user was *not* connected to the Qlik Sense instance. In order to create a continuous time series, the log files had to be upsampled. It was decided to create an binary hourly time series from the logs, where a 1 meant the app was opened during the given our, and 0 meant it was not.

3.4.2 Feature Engineering

The selection and construction of features is an important step since these features will be the input to the models. Therefore it is important to choose relevant data sources and transform them into formats which are appropriate for machine learning.

Visual Inspection

The data was visually inspected in order to find recognizable patterns, such as trends, seasonality, or other cyclical components. In Figure 3.2 the usage pattern for a single app over two weeks is visualized. The y-axis shows how many concurrent sessions are opened, and the x-axis displays the hours of the day with delimiters for when a new day begins.

Due to the relatively short time span the total data covered (just over six months), it was impossible to detect any longer seasonal trends. You would at least need a couple of year's worth of data in order to analyze how the usage pattern changes between for example winter and summer - the data used for this master thesis covered less than one year's worth of app usage. However, shorter trends, such as weekly or bi-daily, could be identified for some apps. Which days had a more prominent usage than other differed between the applications. A usage pattern constant for more or less all apps was the fact that usage was basically nonexistent during weekends.

Cyclic Features

One of the key features derived from the data was a corresponding timestamp to every log entry. The information in these timestamps were subsequently broken down into different features used for fitting the model, such as the hour when an app was opened, which day of the week it was opened, and which day of the month it was opened. These are examples of cyclical ordinal features, and they have to be represented in a way which respects their cyclicity. For example, when using an hour as a feature, there is a very real problem with representing the hour with an integer. Even though the hours 0 and 23 are close to each other in real life, they would be very far away from each other when represented on a linear timeline. To remedy this, these features were evenly mapped onto the unit circle and represented by Cartesian coordinates. This means that values at the beginning and end of a cycle, such as the hours 0 and 23 in the hour of the day cycle, are treated as being close to each other.

A visualization of this can be viewed in Figure 3.3. The hour 12 would therefore be represented by two features; one for the x-axis and one for the y-axis. The day of the week and day of the month was also changed into being represented by Cartesian coordinates.

Lagged Features

A lagged variable is a variable which has its value coming from an earlier point in time, and these variables can be used as features when training a model. Choosing which time lags are

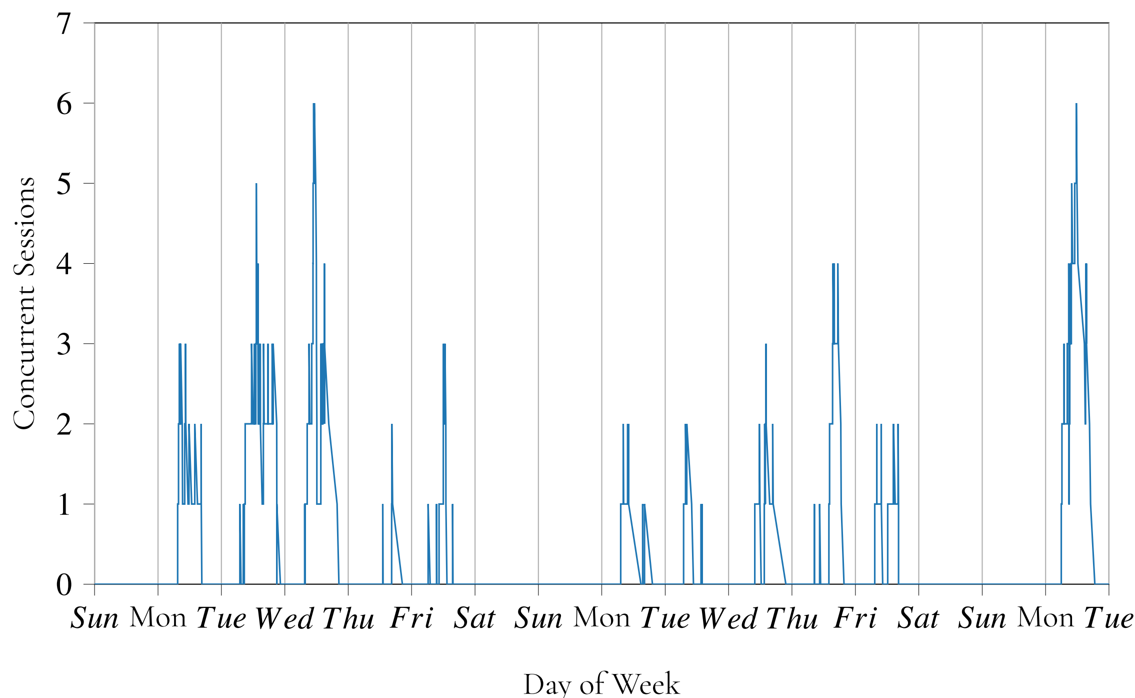


Figure 3.2: The visualized usage pattern over two weeks for one example app. The y-axis shows the number of concurrent sessions at a given time. It is worth noting that there are no recorded session during the weekends for these two weeks.

to be used is important since different lags will exhibit different strengths of correlation. However, this can be done with the autocorrelation function, which is described in detail in Section 2.4.1.

A model can have several lagged features, each with a different time difference. By analysing the usage pattern's autocorrelation, it is possible to either manually or automatically choose the most relevant time lags. An example of an autocorrelation plot for one of the apps is shown in Figure 3.4. The y-axis shows the numerical autocorrelation value, and the x-axis displays the lag in hours. What can be seen in this particular graph, and was also a recurring pattern in several other apps, are the spikes in autocorrelation at and around the lags 24 and 48. Another spike that often occurred was at lag 168, which is how many hours there are in one week. What this means is that a sessions for this particular app often was initiated around the same time every day. However, different apps showed different autocorrelation patterns, which meant it was of utmost importance to actually make use of the autocorrelation function in order to identify the correct time lag for the specific app.

3.5 Evaluation and comparison

Achieving results with a machine learning model is only half of the work. The second part is to actually understand what the results are saying. Hence, it is very important to evaluate the model in a sufficient manner. Since this thesis aims to also compare machine learning models, it is also of interest to identify comparable variables, and then design a satisfactory

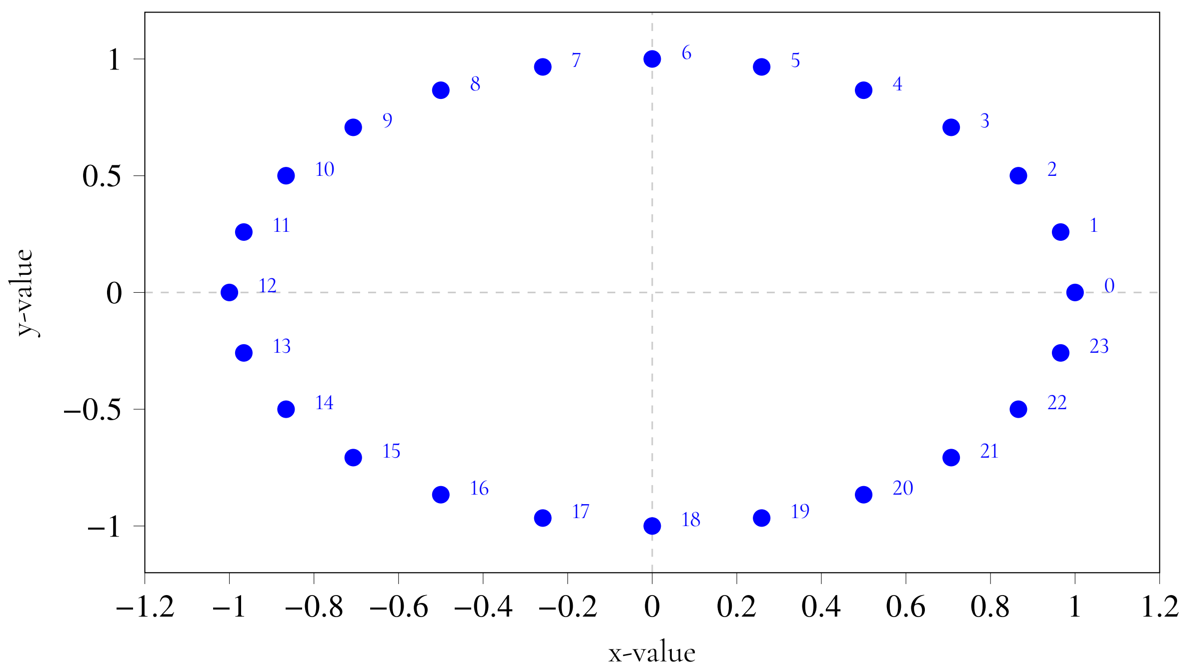


Figure 3.3: The representation of the hours of the day using Cartesian coordinates, when mapped onto the unit circle. Here it can clearly be seen how the hours 00:00 and 23:00 are represented as being close to each other.

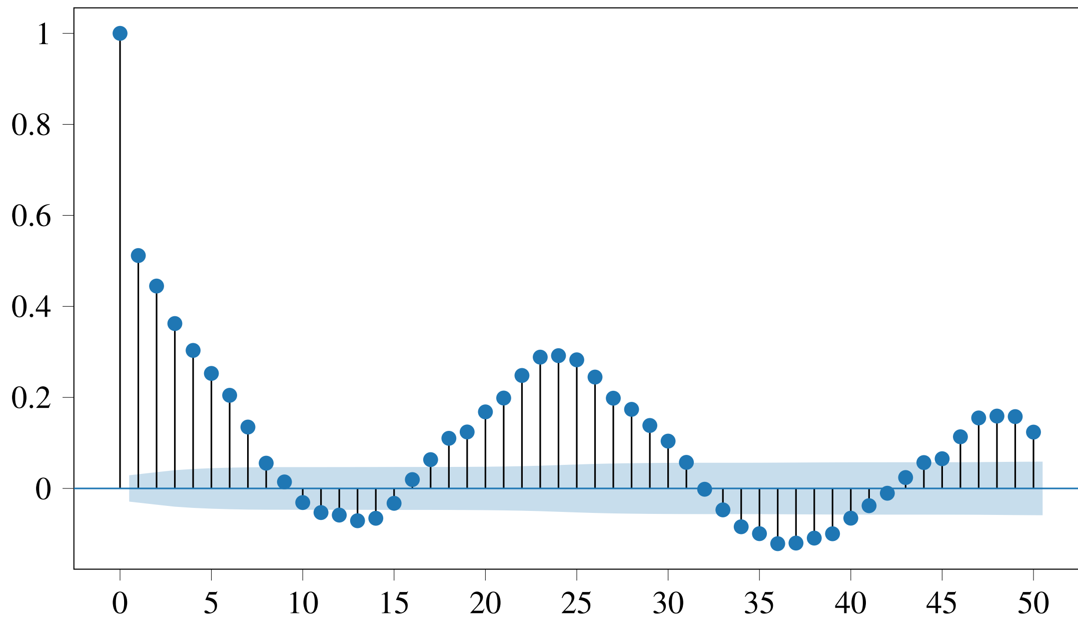


Figure 3.4: A plot for an autocorrelation for 50 time lags for one example app. There are notable spikes around lags 24 and 48. This means that the app is opened about the same time everyday.

way to execute this comparison.

3.5.1 Data simulation

A prediction simulation was conducted in order to imitate how the machine learning would perform in a more real environment. How this simulation works is by predicting the probability for a user opening an app not only for the given hour, but also for each and every of the seven upcoming hours. These eight probabilities are subsequently stored as a row in a matrix. This process is repeated for every hour in the validation set.

The reason as to why eight hours was chosen as the number of hours to calculate a probability over is because, as previously mentioned, this is the time an application is stored in memory.

3.5.2 Model evaluation

To actually make sense of the prediction simulation, an evaluation script was written. This script compares the model predictions to validation data. It does so by summarizing all the values in a row from the probability matrix, and compares this to a previously set threshold. If the accumulated probability is higher than the threshold, the model tries to open the app. The evaluation script then returns what is assumed to be the most relevant features and results during validation:

- *The number of correct predictions.* Correct predictions, in this context, simply means whenever the model correctly predicted a user to open the app within the coming eight hours, which is for how long an app stays in memory.

- *Late openings.* Sometimes the model did predict a correct usage, however it was too late and tried to load the app after it had already been opened.
- *Completely missed app openings.* The times when a user opened the app and the model did not, during the entire time the app was loaded in memory, try to precache it.
- *When the model wrongfully preloads an app.* Despite no user opening the app during the upcoming hours, the model still tried to precache the app.
- *How long time a correctly precached app was stored in memory.* An average of the time before a user actually tried to open an app after it had been precached.

These outputs are subsequently used in different ways in order to concretize and evaluate the model's performance. As described in Section 2.5.1, the first four of these outputs can be visualized in a confusion matrix in order to get a general understanding of how the model performed. The MCC-score, described in Section 2.5.2, can be used to generate a numerical value between -1 and 1 which approximates how well the model is performing. MCC takes into account all four values in the confusion matrix, and a high value (close to 1) means that both classes are predicted well, even if the classes are very unbalanced. The confusion matrix and MCC-score for both models are described in Sections 4.1.1 and 4.2.1 respectively.

3.5.3 Comparison

In order to do a fair evaluation and comparison of the two different machine learning models, several metrics will be weighted together. The Matthews correlation coefficient will, of course, be used to get an understanding of the models' accuracies. However, the accuracy alone is not a sufficient metric to evaluate a model on. Due to the cost of precaching an app, the false positive result in the confusion matrix is the worst predictive outcome a model can generate. This would result in an app taking up unnecessary memory space without a user ever trying to access it. When comparing the models to each other, this outcome therefore outweighs the others, and it is of highest priority to minimize it.

The actual memory usage, both during training and during prediction, is also of interest. A more light weight model is most often preferred, assuming the accuracy is sufficient. This correlates directly with training time - the lower the better. It might be unfeasible to use a model if it takes over one hour to actually fit the model. All of these features will be considered when comparing the models, and they will thereafter act as a basis when motivating the use of one model.

Chapter 4

Implementation

This chapter aims to describe the process of implementing and evaluating the two different machine learning models used for this master thesis, and also the implementation and evaluation of the static rules simulation. Each section starts with an introduction, where the planning phase of the implementation is described. How the implementation was carried out is subsequently described in the subsection thereafter. Lastly, preliminary results are presented, evaluated and discussed. Both models predict the usage pattern of the same app in order to make the evaluations fair, however a general evaluation for the rest of the apps will also be described.

4.1 Feedforward Neural Network

One of the most prominent advantages of FNN is the fact that it is rather easy to implement, has a widespread support in plenty open source libraries, and it has been proven to work many times over when it comes to supervised learning. The problem is that Feedforward Neural Networks are generally used for supervised learning with data that is non-sequential. In order to use FNNs for time-series forecasting, it is possible to use features that describe the current state at time T, which the network will use to predict the state at time $T + 1$.

Several combinations of features where tested in order to see which specific setup generated the results with the highest accuracy. All of these setups were evaluated by using the MCC, and the results are presented in Section 4.1.2.

4.1.1 Implementation

Features

The data available from the log files presented two important feature categories: usage information and timestamps. From these two categories, it is possible to derive many combina-

tions of features that are relevant for detecting trends. From the timestamps, several features were created. The cyclic values used as features were the hours of the day and the day of week. These were described using a cyclic transformation, as described in Section 3.4.2.

Historic usage data was also used as features. A couple of lagged values were automatically selected through using autocorrelation values. Besides this, both a rolling and an expanding mean value were also used as features. The rolling mean value is the overall mean for the given timestamp, while an expanding mean is the mean hitherto calculated in the training process. Lastly, the usage activity for the last 24 and 48 hours were used as two different features.

Hyperparameter Tuning

The optimization algorithm described in Section 2.4.2 was used. It chose and tuned several parameters for the model. The parameters involved and their search space were:

- Loss function (Binary cross entropy, mean square error)
- Activation function (Softmax, tanh, sigmoid)
- Number of hidden layers (0-6)
- Number of neurons in each layer (1-32)
- Learning rate (10^{-6} - 10)
- Dropout rate (0.0 - 0.3)

The loss and activation function choice consisted of trying out several different functions, and then opting for the one that yielded the best result. These two functions are therefore described in the following subsections.

Activation function

The activation function is used to transform the input data of a neuron into an output signal. There are a number of common activation functions in use with artificial neural networks, and the sigmoid function is one of the most used activation functions [32].

The shape of the sigmoid function is that of an S, which can be seen in Figure 4.1. This function will only produce positive numbers between 0 and 1, which makes it the most useful for training data that is also between 0 and 1 [32]. Since the input data used for this master thesis was converted into to a binary string of 1's and 0's, it does seem reasonable that the sigmoid function was deemed the best alternative by the parameter optimizer.

Loss function

The loss function ultimately chosen by the optimization algorithm was the Mean Squared Error (MSE) loss function. This is a common loss function which calculates the mean squared error between the model's predicted value y, and the actual target x [37], and can be calculated the following way:

$$\mathrm{MSE}(\mathbf{y}, \mathbf{t}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - t_i)^2. \tag{4.1}$$

The mean squared error is subsequently passed backwards through the network via back-propagation, as described in Section 2.4.3, and the nodes' weights are updated accordingly.

The MSE has several features desirable for optimization and statistics [37]. First of all, it is very simple. It is inexpensive to compute, and it is also memoryless - meaning that the MSE can be evaluated at each sample, independent of other samples. Lastly, the MSE currently is, and has historically been, widely used. This saves time and effort when implementing, but it is also reassuring knowing that its effectiveness already has been established.

4.1.2 Results and evaluation

Output Data

The model's output data needs to be usable when deciding when to load an application. As described in Section 1.1, the requested information should be output in the format in a list of label points, dictating when certain applications should be loaded. This list is subsequently used as input in the scheduling task provided by Qlik. This means that, roughly speaking, the model is to predict a value for the next time-step. This value should express whether or not the application will be used in the next time-step, in either a binary or numerical format.

When deciding whether or not to load an application, the most important factor to predict is whether or not a user will open the application soon. Since the app is stored in memory for eight hours after a session ends, it is of no interest to predict the degree of usage. More often than not, only the first user that opens an app will be the one who benefits from the precache.

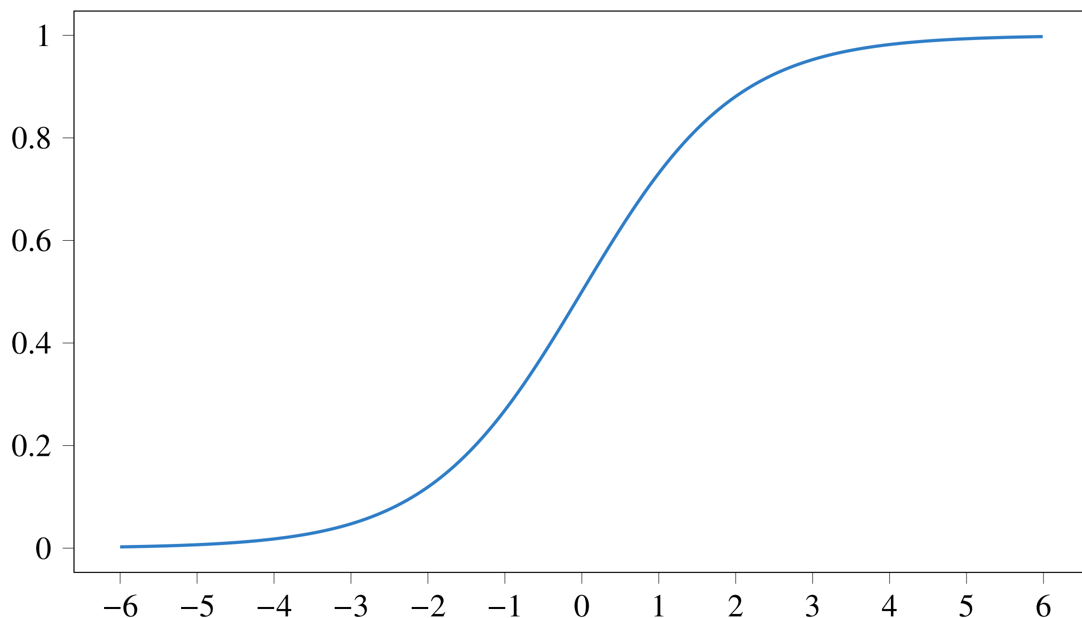


Figure 4.1: A plot of a sigmoid function. As can be seen, the function assumes all values between 0 and 1. This is a commonly used activation function when it comes to binary classification problems.

Evaluation on a single app

Different combinations of features were tested in order to identify which setup yielded the highest prediction accuracy. The results were evaluated using MCC, as described in Section 2.5.2, and a comparison between the different setups can be viewed in Figure 4.1. The highest performing combination of features, according to this comparison, is when using every feature except for the cyclic values.

Table 4.1: The different MCC-scores for the FFN when using different feature combinations. Except from when refraining using means as a feature, all the different setups performed reasonably well.

Features used	MCC Score
All features	0.73
No lags	0.72
No cyclic features	0.76
No means	0.63
No past activities	0.67

Figure 4.2 shows the results from training a simple FNN using the highest performing combination of features, and then simulating it. The data shown here is for two weeks. The yellow line shows the binary input data and displays when a user starts and ends a session; a 1 is an ongoing session and 0 means no session is ongoing. The blue line shows the accumulated probability of a user opening an app sometime during the upcoming hours, as explained in section 3.5.1. The vertical dotted lines represents precaching attempts; a green dotted line means it was successful in precaching, and a red dotted line means a failure. The green background is used to display when an app is stored in memory; a white background means it is currently not stored in memory.

This graph displays just two weeks of the validation data in order to make it easier to study, but there are still some conclusions that can be be drawn. First and foremost, the missed precache was on a Sunday. This is not exactly surprising, seeing how there usually are no openings during weekends, and the overall probability (based on previous lags) therefore is quite low. As discussed in section 2.5.1, the worst case scenario is when the model would try to open an app without a user ever using it. That is why it is preferred to actually miss the weekend openings, instead of having it preloading apps on weekends. Overall, the model does seem to what it is supposed to.

In order to get a more graphical overview of the model's performance, the results were also visualized in a confusion matrix, which can be seen in Figure 4.3. What can be determined from this is the fact that the model did not try to unnecessarily precache an app a single time. This yields an MCC-score of 0.76 which is a good score. What has to be noted is the fact that the app being evaluated here is one of those that had a particularly high usage, which means that it might allow for a higher MCC-score than it would generally score.

Generalised evaluation

In order to get a general understanding of how well the trained model adapts to new apps, it was evaluated against the eleven most used apps in the Qlik Sense instance used for this mas-

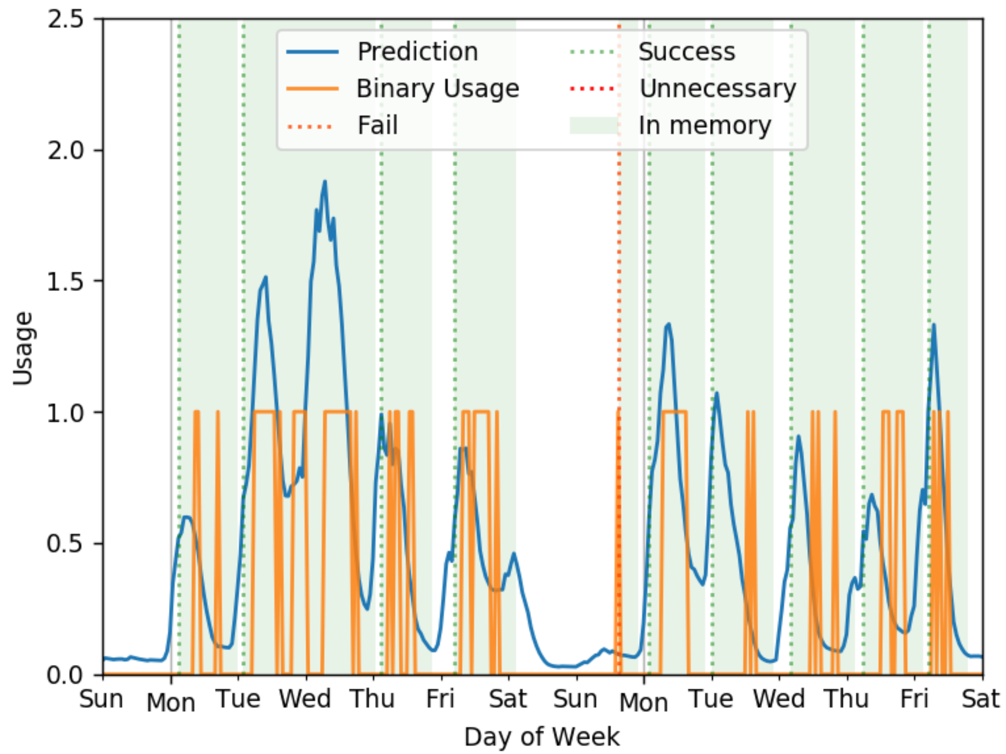


Figure 4.2: A visualization of the simulation of an FNN. This is just an excerpt of two weeks from the entire validation. The green dotted lines represent successful precaches.

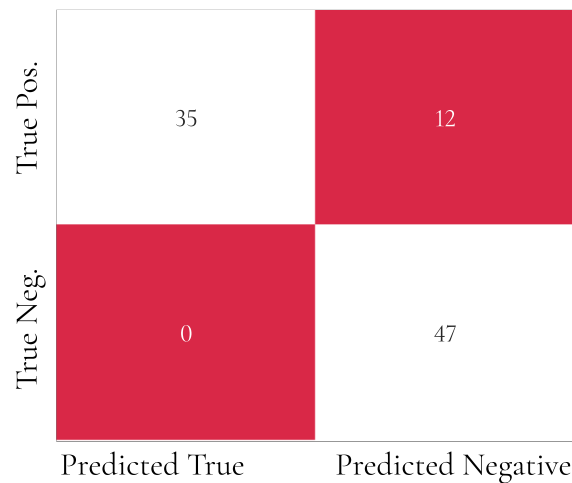


Figure 4.3: The confusion matrix for the simulation of the FNN. In the bottom left and bottom right quadrants are the number of correct predictions. In the top right and bottom left are the failed predictions.

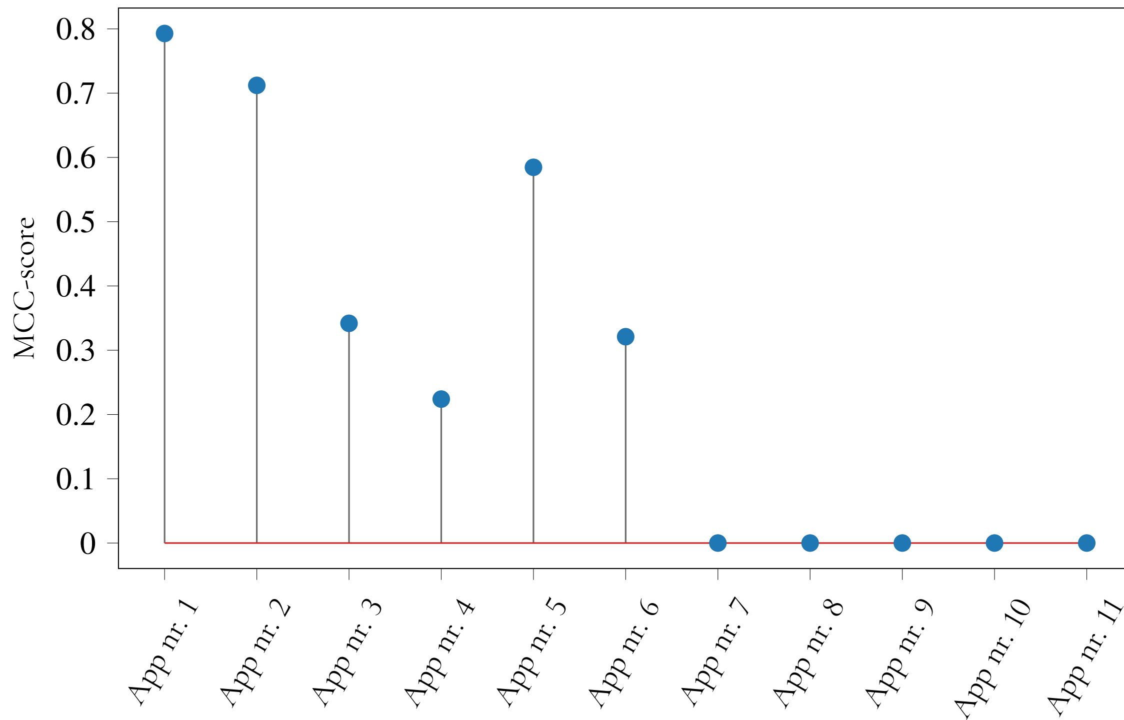


Figure 4.4: All the individual MCC-scores for the different apps. The y-axis represents the MCC-score. The x-axis represents which app is being evaluated.

ter thesis. The complete and more detailed evaluation results can be found in Table 4.2. The individual MCC-score for each and every one of the different apps are visualized in Figure 4.4. What can easily be observed is the fact that the last four apps yielded an underwhelming 0 in MCC-score. Whenever the two diagonals in a confusion matrix are symmetrical, i.e the product of the values in each respective diagonal are equal, the nominator (and consequently the result) in the MCC-equation becomes $\mathbf{0}$. This happens, for example, when both the TP and TN (Success and Fail in Table 4.2) are 0, and is explained more thoroughly in Section 2.5.2. When looking at the evaluation results in Table 4.2, it can be seen that the FNN almost never tries to preload the last four apps. This is due to the very scarce usage pattern of these apps. The consequence of this is that the model's accumulated probability of a user soon opening an app almost never goes above the limit threshold to precache an app at a given timestamp; there simply is not enough data to base the assumption on. This is further described in Section 3.5. This also results in the nominator in the MCC-equation, and therefore also the MCC-score, being 0 for these four apps.

The accumulated confusion matrix, which can be seen in Figure 4.5, does however show some more positive figures. The square in the bottom left is the worst case, i.e the result to minimize. As can be seen, it is indeed very low, which is a very promising result. However, the number of missed precaches, i.e where the model tries to precache an app after it had already been opened, is almost as large as the successful precaches. A vast majority of these, however, come from the apps with below 0.4 in MCC-score; 88 out of 117 failed precaches are from these seven apps. This can most likely be accredited to the fact that the limit threshold

Table 4.2: The evaluation results for FNN. The different columns are the outcome from the evaluation algorithm used, and is further described in Section 3.5.2.

	MCC	Success	Fail	Unnecessary	No activity	Time in cache (hours)
App nr. 1	0.79	44	13	0	57	1.07
App nr. 2	0.73	32	6	6	47	1.03
App nr. 3	0.36	13	19	4	37	2.0
App nr. 4	0.26	6	32	1	52	3.25
App nr. 5	0.49	21	13	8	47	1.82
App nr. 6	0.32	7	21	1	31	1.25
App nr. 7	0.0	0	7	0	45	N/A
App nr. 8	0.0	0	0	0	129	N/A
App nr. 9	0.0	0	8	0	9	N/A
App nr. 10	0.0	0	0	0	13	N/A
App nr. 11	0.0	0	1	0	94	N/A

to precache an app is deliberately set a bit high. This is done in order to assume a more pessimistic approach to precaching and consequently minimize the number of unnecessary precaches. If the model was to be re-simulated with a lower threshold, the number of failed precaches would be lowered, while both the number of successful and unnecessary precaches would increase due to the more optimistic approach to precaching.

This pessimistic approach can also be viewed in how long the apps are loaded in the cache before a user opens the app. Since the data used to train the models was sampled to hours, the shortest amount of time an app can spend in cache is one hour. One hour in cache means that the app was precached the timestamp before a user opened the app, due to how the simulation is coded. Looking at Table 4.2, it can be noted that the time spent in cache is overall very low; most often closer to one hour.

The approach taken when training the FNN therefore results in the model trying to precache an app as close to a real opening as possible. This has the deliberate consequence that there exists an overrepresentation of failed precaches due to the pre-established model requirements, where it is preferable to miss precaches in order to minimize the number of unnecessary precaches.

4.2 Temporal Convolutional Network

In comparison to FNN, the TCN is relatively new and therefore lack any official implementation in the most popular machine learning libraries such as Tensorflow, Keras and PyTorch. Therefore, it was decided to use a third-party implementation by Philippe Rémy that is implemented using Keras [30]. The main practical difference when using a TCN instead of an FNN is that the data input and outputs are sequences of data. Whereas an FNN might have an input of a single hour's features, a TCN can have an input of a sequence of several hours, each with their own features.

	Predicted True	Predicted Negative
True Pos.	126	117
True Neg.	23	556

Figure 4.5: The accumulated confusion matrix for all evaluations when using an FNN. This is basically every result from Table 4.2 summed together in a confusion matrix in order to produce a result more easy to understand.

4.2.1 Implementation

Data

Whereas the FNN had features describing a single time point, a TCN can handle a sequence of time points, each with their own features. The number of inputs to the TCN is therefore the number of features per time point multiplied with the sequence length. Thus, some features (such as lagged values) used in the FNN might not be as useful in a TCN since this information is captured in the sequence instead.

Similarly to the TCN's input, the output is also a sequence. However, this output can also be a sequence of length one, meaning that the network can be used to predict only a single value. For this problem domain, there are two possible approaches in terms of output data. The first approach is to train a network that can predict usage for several coming hours. The second approach is to predict only one hour ahead by having a single output value. This model can then be used to predict several hours ahead by iteratively use the model's output to create a new input.

Hyperparameter Tuning

The hyperparameters of the TCN model was tuned using the method described in Section 2.4.2. Compared to the FNN model, the TCN model had a higher amount of hyperparameters to tune and generally took longer to train. This resulted in the hyperparameter tuning step taking considerably longer time for this model. The parameters subject to automatic tuning are listed below along with the search space.

- Loss function (Binary crossentropy, mean squared error)
- Activation function (Softmax, tanh, sigmoid)

- Number of filters (4 - 32)
- Size of kernel (1 - 16)
- Number of dilations (5 - 16)
- Number of stacks (1 - 3)
- Dropout rate (0.0 - 0.3)
- Sequence length (48, 168)

Both the loss and activation function chosen by the hyperparameter optimizer were the same as for the FNN, i.e. the mean squared error (MSE) as a loss function, and the sigmoid function as an activation function. Both of these functions are, as previously stated, some of the most common functions within their respective areas. That they are flexible enough to be applicable for two different models but for the same problem scope is therefore not especially surprising.

4.2.2 Results and evaluation

Evaluation on a single app

Just as with the FNN, different combinations of features were evaluated in order to identify which setup yielded the best result. The results from this comparison can be viewed in Table 4.3. Once again, just as with the FNN, is the best performing combination of features when using every feature except for the cyclic values.

Table 4.3: The different MCC-scores for the TCN when using different feature combinations. The results have some obvious highs, and some obvious lows, where using every feature except for the cyclic features are the best performing feature combination.

Features used	MCC Score
All features	0.73
No lags	0.58
No cyclic features	0.75
No means	0.69
No past activities	0.70

The results were also visualized in a confusion matrix, as can be seen in Figure 4.6. The model successfully manages to precache a majority of app openings, while at the same time not preloading an app unnecessarily one single time. This results in an MCC-score of 0.75, which indicates that the classifier is performing well.

	Predicted True	Predicted Negative
True Pos.	16	9
True Neg.	0	30

Figure 4.6: The confusion matrix for the simulation of the TCN. In the bottom left and bottom right quadrants are the number of correct predictions. In the top right and bottom left are the failed predictions.

Generalised evaluation

The MCC-scores, when looking at the performance for the different apps in Figure 4.7, are overall looking to be either very high or very low. The evaluation for the last four apps once again yield a 0 in MCC-score. This can most likely be accredited to an overall low usage for these specific apps.

The results are described in more detail in Table 4.4, and are summarized in a confusion matrix, Figure 4.8. What can be viewed here is that the number of unnecessary precaches, which is supposed to be minimized, almost equals the number of failed precaches in numbers. While this is alarming by itself, almost half of all the unnecessary precaches, 48%, comes from the apps that scored an MCC-score below 0.2.

Studying the detailed evaluation results closer, it can noted that with two exceptions (the apps scoring below 0.2 in MCC-score), the TCN did actually yield a higher amount of unnecessary precaches than failed precaches. This might be due to the limit threshold when simulating being set too low.

As a final note it is worth mentioning that the time in cache is very low. As previously stated, the shortest amount of time an app can spend in cache is one hour. Seeing therefore how every app, including those below 0.2 in MCC-score, are stored in cache below two hours is impressive.

4.3 Static Rules

The static rules are app openings manually scheduled by an administrator, as described in Section 1.1. These rules would most likely be different for each of the apps in a Qlik Sense instance due to different usage patterns. This is assuming the static rules are at all implemented, since scheduling app openings is an optional setting which not everyone uses. Considering how much time and effort one could devote to optimizing the manually scheduled preloads,

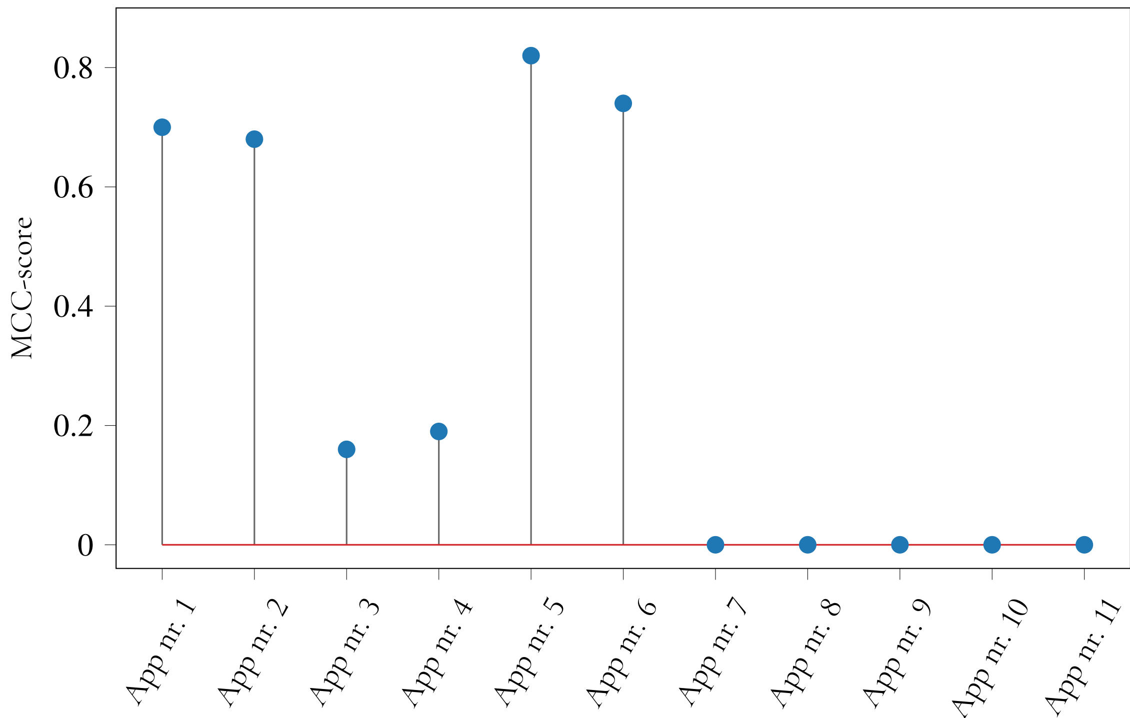


Figure 4.7: The MCC for all apps when using the TCN. The y-axis represents the MCC-score. The x-axis represents which app is being evaluated.

Table 4.4: The evaluation results for the TCN. The different columns are the outcome from the evaluation algorithm used, and is further described in Section 3.5.2.

	MCC	Success	Fail	Unnecessary	No activity	Time in cache (hours)
App nr. 1	0.70	16	9	0	30	1.81
App nr. 2	0.68	12	0	8	27	1.58
App nr. 3	0.16	6	7	11	27	1.17
App nr. 4	0.19	5	13	4	27	1.60
App nr. 5	0.82	11	0	3	19	1.00
App nr. 6	0.74	14	1	5	26	1.14
App nr. 7	0.00	0	2	0	16	N/A
App nr. 8	0.00	0	0	0	1	N/A
App nr. 9	0.00	0	0	0	1	N/A
App nr. 10	0.00	0	0	0	1	N/A
App nr. 112	0.00	0	0	0	1	N/A

it is impossible to say what degree of accuracy to aim for when simulating these static rules.

The purpose of this implementation, which is one of the most simplistic approaches possible, is to compare it to the other two machine learning models. This is done to get a more thorough understanding of how well the two machine learning models are performing when put into perspective. The assumption made when simulating the static rules is therefore that every app is scheduled to open the same time of the day, every workday of the week. While this might not be an entirely truthful representation of how the rules are actually decided upon in a production environment, it was the most reasonable assumption to make with regards for the time constraints of the master thesis.

4.3.1 Implementation

In order to produce usable results for the master thesis, a generator for static rules was created. This program had three input parameters: the data for the app currently being evaluated, the timestamp for when openings were to be scheduled, and the length of the validation data chunk. The same app data was used for preliminary evaluation for the both machine learning models, and also this static rules simulation. This meant that even though the output from this generator was nothing but a list of timestamps for when the app were to be scheduled, it had to be of the same length as the validation data used for the other two models in order to be comparable.

The output data was subsequently evaluated through the evaluation script, as described in Section 3.5, and thereafter visualized. Since there was no parameter optimization or feature engineering of any sort, it was very fast to try and evaluate different timestamps for the app openings. This trial and error-approach was also automated in order to more easily be able to adapt the rules to new apps. What is being presented in the following subsection is therefore the timestamp which yielded the best MCC-score.

Figure 4.8: The accumulated confusion matrix when using TCN as a model. This is basically every result from Table 4.4 summed together in a confusion matrix in order to produce a result more easy to understand.

4.3.2 Results and evaluation

Evaluation on a single app

The best result gained was when the app openings were scheduled to be executed at five o'clock in the morning. This seemed to be a optimal spot for when it caught most of the early openings, but at the same time also caught most of the late morning openings. Figure 4.9 visualizes a two week span for when the scheduled openings were evaluated. As can be seen, the opening on a Sunday is missed. This is to be expected, since the scheduling is only for workdays, i.e Monday up to and including Friday. As opposed to the other simulation visualizations, this graph only shows the binary usage pattern together with app openings. This is because of the scheduling not actually having an accumulated probability; it simply opens the app the same time every day regardless of the circumstances.

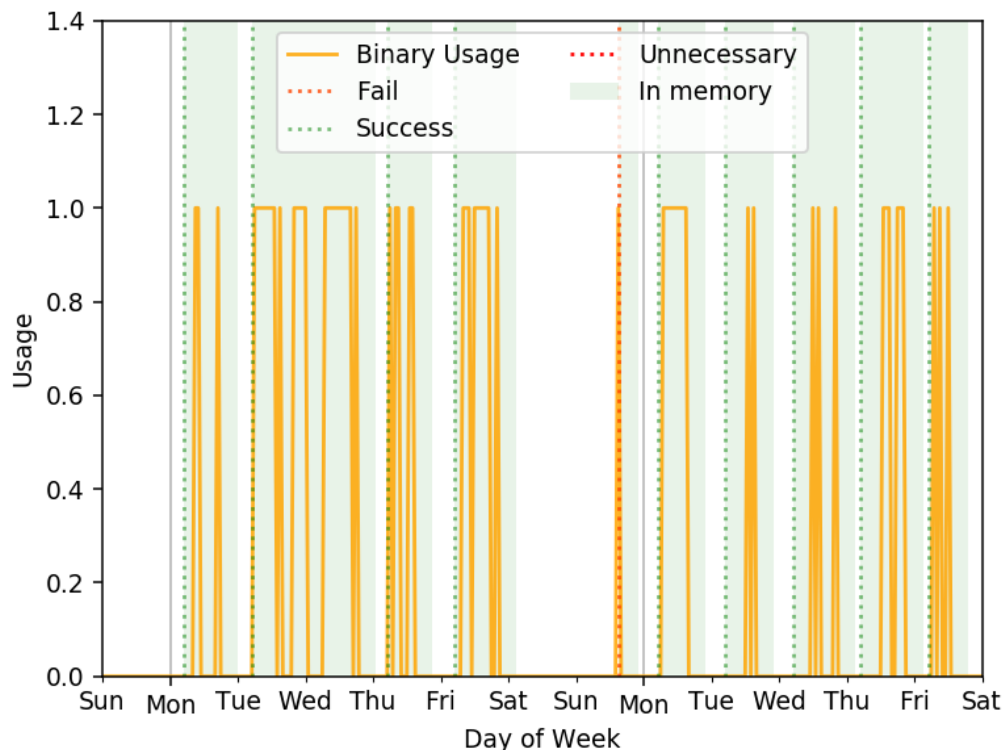


Figure 4.9: The visualization of the simulation for the static rules. Once again, this is merely an excerpt of two weeks from the entire validation data set. The precaches are hard coded as timestamps when programmed as static rules, this is the reason why there is no blue line visualizing the model's predicted probability of there being an opening.

All in all, the static rules did perform better than expected. Looking at the confusion matrix in Figure 4.10, it managed to successfully preload an app 26 times, while missing 19 openings. Even though the app was opened twice without a user ever accessing it, it is not

unreasonable to assume that this would have happened during days with low usage frequency. An opening would most likely be scheduled only during days with known high usage, meaning it is not guaranteed these kinds of misses would occur in a production environment. What this results in is an MCC-score of 0.61, which is still quite high in the positive result spectrum. However, it has to be noted that the average time in cache before the first opening was 2.97 hours, which is quite high.

	Predicted True	Predicted Negative
True Pos.	26	19
True Neg.	2	46

Figure 4.10: The confusion matrix for static rules for a given app. In the bottom left and bottom right quadrants are the number of correct predictions. In the top right and bottom left are the failed predictions.

Generalised evaluation

The MCC-scores for the static rules seem to be quite scattered while also ignoring the usage amount, when studying the graph in Figure 4.11. What may be most surprising is how high the static rules actually manages to score on some of the apps, with a maximum MCC-score of 0.85 when applied to App number 5. This is mostly like due to a quite constant and predictable usage pattern, where the app is opened at approximately the same time every day.

Looking at the accumulated confusion matrix in Figure 4.12, it can be noted that it is pretty much the opposite of the generalised results from FNN. The static results actually have fewer failed precaches, but at the cost of more unnecessary precaches. This is not a particularly wanted behaviour since the unnecessary precaches are what is meant to be minimized.

The Table 4.5 gives some more detailed information about the evaluation results. What can be noted here is that almost all apps have the result from the evaluation spread out in all four columns; this results in a non-zero nominator in the MCC-equation (Equation 2.3) and consequently in the MCC-score not being 0. A non-zero MCC-score in this context does not mean that static rules is necessarily a better approach, it just means static rules generates a larger amount of precaches.

These quite scattered MCC-results further proves that in order to utilize these static rules in an efficient manner, rigid manual usage pattern analysis has to be conducted in order to

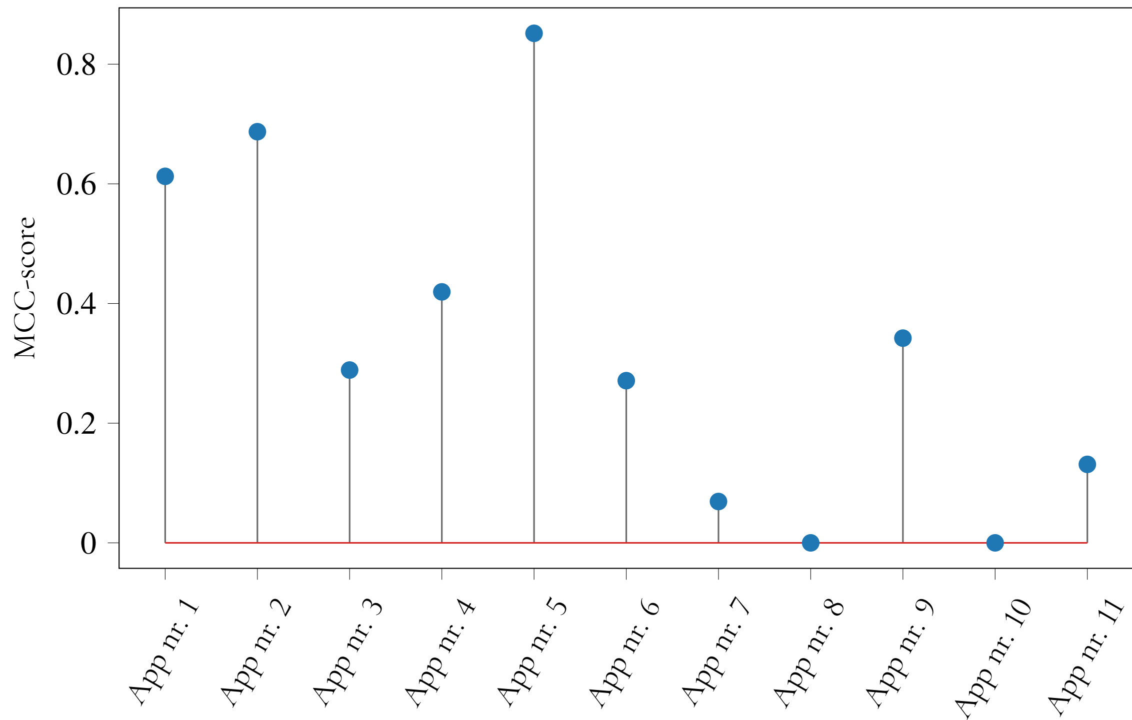


Figure 4.11: The MCC when implementing static rules for the different apps. The y-axis represents the MCC-score. The x-axis represents which app is being evaluated.

identify the best time to preload an app every weekday. This can most likely be linked to the fact that few of the apps exhibit a constant similar behaviour every weekday, which can be seen in the large amount of unnecessary precaches.

Figure 4.12: The accumulated confusion matrix for all apps when using static rules. This is basically every result from Table 4.5 summed together in a confusion matrix in order to produce a result more easy to understand.

Table 4.5: The evaluation results for static rules. The different columns are the outcome from the evaluation algorithm used, and is further described in Section 3.5.2.

	MCC	Success	Fail	Unnecessary	No activity	Time in cache (hours)
App nr. 1	0.61	36	22	3	60	2.97
App nr. 2	0.69	34	4	12	52	2.79
App nr. 3	0.29	19	13	23	58	5.26
App nr. 4	0.42	24	14	14	52	4.71
App nr. 5	0.85	32	2	4	44	2.97
App nr. 6	0.27	17	11	24	54	4.47
App nr. 7	0.069	4	3	41	53	6.0
App nr. 8	0.0	0	0	31	33	N/A
App nr. 9	0.34	5	3	5	14	3.2
App nr. 10	0.0	0	0	1	2	N/A
App nr. 11	0.13	1	0	31	38	6.0

Chapter 5

Results

This chapter aims to go through the results gained in the previous chapter. Firstly, a summary of the results for each model will be presented, followed by a short discussion. These summaries will thereafter be compared to each other in the following section. Based on this comparison, a couple of different conclusions can be drawn.

5.1 Evaluation

This section presents a summary of the results gained from each respective model, and from the static rules. A brief discussion follows the results, where potential causes for the results are contemplated. It is also discussed whether or not the model actually performed well, when compared to the model requirement stated earlier in the thesis.

5.1.1 FNN

The overall result of the evaluation for FNN is that the FNN is a rather lightweight neural network that is easy and fast to train (the FNN only took 47.4 ± 11.1 seconds to train), while still yielding a quite high prediction accuracy. The full results for the training times can be viewed in Table 1 in the appendix. It rarely precaches an app unnecessarily, but rather prefers to either wait and do nothing, or tries to precache the app too late. However, it does not fare well when an app exhibits a low usage or a more complex usage pattern. As can be seen in Table 4.2, it does miss a lot of app openings while the amount of usage is still quite high (specifically app number 3, 4, and 6).

Regardless, considering how fast and painless it is to optimize and train this model it is still a very well performing network. The apps that do become precached are done so quite close to the actual app opening, which is a wanted behaviour.

5.1.2 TCN

TCN took a long time to optimize and train (the TCN took 229.8 ± 36.2 seconds to train), and it was quite taxing on the hardware while training. The full results for the training times can be viewed in Table 2 in the appendix. This is simply a sophisticated and quite heavy network, which might be built for more complex problems than the one at hand. This is especially true when considering the relatively short time span the data stretches over; the complexity and robustness of the model might be better suited for longer time series.

When studying the detailed evaluation results in Table 4.4 it can be noted that TCN's results did seem to a bit of hit-or-miss, regardless of usage amount. However when it did perform well, it did so exceptionally. This might be because of TCN being more sensitive to hyperparameter optimization than FNN, or maybe it is harder to create a generalized model based on this network.

5.1.3 Static rules

The static rules evaluated in this master thesis did perform better than expected, while still not performing especially well. Looking at Table 4.5 it can be noted that the relatively high MCC-scores can be accredited to the fact that the static rules simply precaches very often. This, combined with a bit of luck, results in good results.

The results are the worst when the actual app usage is low, since it keeps precaching apps even though no one is using them, and the best when app usage is high. This comes as no surprise, but it should be reiterated that these manual rules in a production environment are preceded by a (hopefully) rigid usage pattern analysis. This makes the rules used in this master thesis look a bit extra static.

5.2 Comparison

All in all, the three approaches each have their own advantages and weaknesses. FNN, to begin with, does excel compared to the other two approaches when used together with an app that exhibits a medium to high usage frequency, a quite clear usage pattern, and when it is not required to have an extraordinarily high prediction accuracy. This is due to the fact that the FNN, the way implemented in this master thesis, prefers to deliberately fail precaches in order to minimize the number or unnecessary precaches. Coupled with the fact that the FNN is faster to train and overall more lightweight compared to the TCN gives the FNN the clear advantage when optimizing memory usage and computing capacity is a priority rather than pure precaching accuracy.

The TCN, on the other hand, has a better prediction accuracy, but at the cost of more unnecessary precaches. It is also harder and more computationally expensive to optimize and train. TCN therefore exceeds when an app exists that exhibit a medium to high usage frequency, and achieving a high prediction accuracy is what is sought after.

The static rules is a fully viable option compared to the other two models. A manual rule set overshadows both of the machine learning models when it comes to an app with a very low usage frequency with a given pattern, but it is of utmost importance to precache the few openings that exists. For example, if an app is only opened the first Monday every month, it

would take the two machine learning models at least over a year's worth of data to recognize this pattern. A manual rule set, however, could be put in place while still achieving a good score. Outside of this usage scenario, i.e. when the app exhibits a natural usage pattern, the static rules are very hard to tune in order to match the prediction accuracy of the machine learning models. That is not to say it is impossible though; if the manual usage pattern analysis is thorough enough it may very well be feasible to match or even perform better than the machine learning models. The question that arises at that point is if it is worth dedicating all that time to manually create a tailored rule set when using machine learning is an option.

Chapter 6

Discussion

The discussion in this chapter is divided into three parts. First is a brief overall discussion regarding things that might have affected the results. Areas discussed are the implementation process, the generalization of the models and the feature extraction. Secondly, the limitations are discussed. This is things that might have affected or influenced the interpretation of the outcome. Lastly, the internal and external validity of the study is discussed.

6.1 Results discussion

The implementation process for the TCN and FNN were very similar in numerous aspects, especially in the steps taken to produce a model. A major distinction, however, is that FNN is a well established model while TCN is comparatively new. Consequently, FNN is more accessible due to the higher number of official platform implementations, publications, community support, and documentation. As mentioned previously, the TCN used was implemented by an individual and it may therefore lack the same level of quality compared to a hypothetical official Keras TCN. Due to limited time, a rigorous analysis and evaluation of the TCN implementation was not done so there might exist flaws in the TCN used. Furthermore, future implementations of a TCN may provide better results in this application.

Another important point to take into consideration is the features used as they can evidently impact the performance of the model greatly. A few features were implemented and tested but there are many that could have been used. Some features may even negatively impact the results, such as the cyclic features. While it is entirely possible to spend more time attempting to create new features, this would require many iterations of optimizing the new models and testing them. Similarly to the hyperparameter optimization and the machine learning field in general, it is possible to improve performance by utilizing more time and resources. However, due to diminishing returns it may no longer be feasible to continue improvement.

The different models' inability to generalize well to other applications suggests that some

optimization needs to be performed for each application. In this case, an FNN is ideal due to its ability to produce sufficient results with limited time and resources. The poor generalization may be a result of certain applications having a higher degree of randomness, or the features used may not sufficiently capture the user trends.

6.2 Limitations

The work in this thesis is based on analyzing and testing data from Qlik's Research and development (R&D) department's production environment. The data is extracted from log files, and stretches over a time span from October up to and including April. While examining these log files, it was discovered that a vast majority of the data was unusable; the raw data consisted of a large amount of different apps, with very few initiated sessions each. Thus, just over 99% of the apps had to be removed when parsing the raw data in order to only retain the apps with a sufficient amount of session starts. It was not a copious amount of data left after this purge, and the apps left with the least amount of sessions were borderline cases regarding if the machine learning models could actually learn from the very sparse usage pattern.

When working with time series, the underlying data relationships working as basis for decision making may very well change drastically over time. This phenomenon is known as concept drift [35], and is a well known problem within the machine learning field. This is something that happened with the app used as benchmark testing in the results sections; the underlying functionality of the app, and consequently the usage pattern, was changed over night, thus invalidating the previous data analysis done on this specific app. As a consequence, it was decided upon stopping the extraction of the logs and instead settle for the data accumulated up to that point. This resulted in the loss of about a month's worth of logs. However, in order to not let the predictive accuracy deteriorate over time, concept drift is something that always has to be worked around in a real production environment.

Another limitation in this application of machine learning is that the MCC-score does not sufficiently encapsulate the business requirements in the sense that it does not take into consider the trade-offs between caching costs, time saved, and similar factors. An important limitation to using the MCC-score is that it does not provide any insight into the trade-off between unnecessary openings and failing to precache. This means that comparing models using only the MCC-score does not accurately compare the different models' ability to precache according to the business requirements.

Furthermore, the data is to some degree random which means that it is impossible to accurately predict the usage. This may in some cases result in some models being tuned to maximize the MCC-score according to the random data and the model therefore not performing optimally if it would be used to predict a data set with a lower degree of randomness. This limitation is not caused by the choice of using the MCC-score, but rather the fact that the data has a degree of chaotic randomness that makes it difficult to separate usages that are predictable from those that are unpredictable.

A very important factor when optimizing machine learning models is the amount of processing power available. Since training neural networks is quite computationally expensive, and there exist many possible combinations of hyperparameters, the results can oftentimes be limited by the available processing power. This was especially true for the TCN on account of the relatively long training times and large number of hyperparameters. Due to the

limited time and hardware, the hyperparameters chosen might not be optimal even though it was the best combination available in the search space.

6.3 Internal and external validity

Internal and external validity are two standards of rigor when it comes to research [4]. The internal validity is the question whether or not the observed results are actually caused by the conducted experiments rather than an external influence (or just plain randomness). This is a very important part of scientific studies since internal validity depends largely on the procedures and how well the study is performed. However, while rigorous research methods can ensure internal validity, external validity, on the other hand, may be limited by these methods. External validity is namely the degree to which results of the experiment would generalize to contexts other than those of the experimental conditions. So while internal validity relates to how well a study is conducted (its structure), external validity relates to how applicable the findings are to the real world.

The approach with using cross validation within this master thesis ensures a certain level of internal validity. The results were evaluated using MCC after every training iteration, and it could subsequently be observed that the model became more accurate through training. Besides this, the hyperparameters were optimized in order to identify the combination which generated the highest prediction accuracy. This means that different parameter setups generated different results. When comparing to the static rules in place, it could also be observed that the machine learning models generally produced a higher accuracy. Lastly, different combinations of features were evaluated, and it was also observed here that different combinations yielded different results. From this the conclusion can be drawn that the models actually improved their prediction accuracy from the feature extraction, training, and hyperparameter optimization. This, in turn, shows that the results from the evaluations are not random, but are indeed influenced by the model setup and chosen features.

The purpose of the models trained in this master thesis is that they are supposed to be used as a precaching feature, or extension, for a given instance of Qlik Sense. This means that the models have to be generalisable, i.e exhibit a high external validity. The hyperparameter optimization done in this master thesis is relatively generalised, which means the parameter setup chosen in the end were those that performed rather well on several apps. In order to examine this closer, an evaluation over all apps were conducted. This showed that the models trained were able to adapt to different apps while still performing rather well. In order to achieve the highest possible result, however, it would be preferable to optimize the hyperparameters for each individual app.

Regardless, the approach taken during this master thesis, both during implementation and evaluation, is also applicable to software environments outside that of Qlik. This is provided that the input data consists of log files. The evaluation conducted, however, is a general approach for binary classification problems within supervised machine learning.

Chapter 7
Conclusion

It has been shown in this master thesis that it is indeed possible to create a machine learning model that can predict user behaviour in order to prefetch cloud applications. This has been done through implementing two different networks: a feedforward recurrent network (FNN) and a temporal convolutional network (TCN). These two networks have subsequently been compared to each other. The results from the evaluation can bee seen in Sections 4.1.2 and 4.2.2, together with Tables 4.2 and 4.4 in the appendix. This shows that the the two networks' prediction accuracy is good, albeit not optimal. This mostly applies to apps with a rather high amount of usage; both of the networks struggle to predict accurately when they had little data to train on.

However, what degree of prediction accuracy to aim for depends on how important the precaches actually are. Generally speaking, according to the results in this master thesis, a higher prediction accuracy comes with a higher memory and computational cost. If a high prediction accurcacy is preferable regardless of memory constraints, then a TCN would be the network that performs the best. If a more lightweight approach is sought after, then an FNN would be the best approach. The comparison also shows that the machine learning models overall outperformed the static rules when they actually had enough data to train on, but that static rules are preferable when dealing with a very low usage frequency with a given pattern.

To increase the accuracy of the models, an automatic optimization algorithm has been utilized in order to identify which setup of hyperparameters yields the best result for the two respective models. Different combinations of features were also tested to in order to find the highest performing feature combination. In Tables 4.1 and 4.3 the different tested feature combinations and the corresponding MCC-score can be viewed.

None of the approaches seem to be completely generalisable when considered the approach taken in this master thesis. It would therefore be recommended to use a model as feature toggle where the model is optimized and tailored for a specific app. Both regarding hyperparameters and feature combination.

There is still major work to implement this is in production at Qlik. There will most

likely be technical challenges in the integration, and model improvements would be necessary. However, we are hopeful that our work will be of use reduce computation costs in cloud infrastructure.

Future work

In this master thesis only two types of neural networks have been implemented: FNN and TCN. As mentioned in Section 2.3.2, it is actually the recurrent neural network (RNN) which has been used as the default approach for sequence modeling. It would be highly interesting to see how an RNN would fare compared to the TCN and FNN in this regard.

A different way of acquiring data might also be worth looking into. This master thesis has scraped locally created log files in order to create a trainable dataset. Locally stored log files have the unfortunate disadvantage of sometimes being deleted in order to create more space, if memory is a critical constraint. Some kind of remote logging, where the information is stored on Qlik's end instead might therefore be of interest.

Increasing the size of the data set is another straightforward way to improve the result. It has been a relatively small dataset used in this master thesis, especially when divided into every individual app. To actually collect data for a longer period of time before trying to train a machine learning model might yield better results.

References

[1] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[2] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[3] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control.* John Wiley & Sons, 2015.

[4] Robert O Briggs and Gerhard Schwabe. On expanding the scope of design science in is research. In *International conference on design science research in information systems*, pages 92–106. Springer, 2011.

[5] Matias Callara and Patrice Wira. A probabilistic learning approach for predicting application launches in cloud computing architectures. In *2019 IEEE/SICE International Symposium on System Integration (SII)*, pages 584–589. IEEE, 2019.

[6] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542, 2009.

[7] Davide Chicco. Ten quick tips for machine learning in computational biology. *BioData mining*, 10(1):35, 2017.

[8] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21(1):6, 2020.

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[10] Dan Claudiu Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[11] Emelie Engström, Margaret-Anne Storey, Per Runeson, Martin Höst, and Maria Teresa Baldassarre. A review of software engineering research from a design science perspective. *arXiv preprint arXiv:1904.12742*, 2019.

[12] Behdad Esfahbod. Preload — an adaptive prefetching daemon. 2006.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[14] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[15] Tim Head, MechCoder, Gilles Louppe, Iaroslav Shcherbatyi, fcharras, Zé Vinícius, cmmalone, Christopher Schröder, nel215, Nuno Campos, Todd Young, Stefano Cereda, Thomas Fan, rene rex, Kejia (KJ) Shi, Justus Schwabedal, carlosdanielcsantos, Hvass-Labs, Mikhail Pak, SoManyUsernamesTaken, Fred Callaway, Loïc Estève, Lilian Besson, Mehdi Cherti, Karlson Pfannschmidt, Fabian Linzberger, Christophe Cauet, Anna Gut, Andreas Mueller, and Alexander Fabisch. scikit-optimize/scikit-optimize: v0.5.2, March 2018.

[16] Rob J Hyndman. Measuring forecast accuracy. *Business forecasting: Practical problems and solutions*, pages 177–183, 2014.

[17] Juhani Iivari and John R Venable. Action research and design science research-seemingly similar but decisively dissimilar. 2009.

[18] Nikolay Laptev, Jason Yosinski, Li Erran Li, and Slawek Smyl. Time-series extreme event forecasting with neural networks at uber. In *International Conference on Machine Learning*, volume 34, pages 1–5, 2017.

[19] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. Machine learning-based prefetch optimization for data center applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10, 2009.

[20] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[21] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[22] Tom Michael Mitchell. *The discipline of machine learning*, volume 9. Carnegie Mellon University, School of Computer Science, Machine Learning …, 2006.

[23] Douglas C Montgomery, Cheryl L Jennings, and Murat Kulahci. *Introduction to time series analysis and forecasting*. John Wiley & Sons, 2015.

[24] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *ArXiv e-prints*, 11 2015.

[25] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[26] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[27] Qlik Solutions. *Pre-caching using the Qlik Sense Scalability Tools (.NET SDK).* `https://community.qlik.com/t5/Qlik-Scalability/Pre-caching-using-the-Qlik-Sense-Scalability-Tools-NET-SDK/gpm-p/1477822`, 2016. [Online: accessed 25- May- 2020].

[28] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[29] Mark Russinovich. Inside the windows vista kernel: Part 3. *Microsoft TechNet Magazine*, 2007.

[30] Philippe Rémy. Keras-tcn. `https://github.com/philipperemy/keras-tcn/`, 2018.

[31] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128*, 2014.

[32] P Sibi, S Allwyn Jones, and P Siddarth. Analysis of different activation functions using back propagation neural networks. *Journal of Theoretical and Applied Information Technology*, 47(3):1264–1268, 2013.

[33] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[34] Hokwon Song, Changwoo Min, Jeehong Kim, and Young Ik Eom. Usage pattern-based prefetching: quick application launch on mobile devices. In *International Conference on Computational Science and Its Applications*, pages 227–237. Springer, 2012.

[35] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2):58, 2004.

[36] Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *CoRR*, abs/1601.06759, 2016.

[37] Zhou Wang and Alan C Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE signal processing magazine*, 26(1):98–117, 2009.

[38] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of CNN and RNN for natural language processing. *CoRR*, abs/1702.01923, 2017.

[39] Lingxue Zhu and Nikolay Laptev. Deep and confident prediction for time series at uber. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 103–110. IEEE, 2017.

Appendices

Table 1: Training time for 25 FNNs.

Network	Training Time (s)
FNN-0	59.70
FNN-1	56.16
FNN-2	43.72
FNN-3	31.50
FNN-4	40.96
FNN-5	40.78
FNN-6	37.91
FNN-7	45.68
FNN-8	27.93
FNN-9	66.67
FNN-10	37.76
FNN-11	35.60
FNN-12	42.14
FNN-13	49.31
FNN-14	50.06
FNN-15	56.94
FNN-16	33.64
FNN-17	40.69
FNN-18	71.45
FNN-19	62.22
FNN-20	47.97
FNN-21	42.54
FNN-22	46.84
FNN-23	56.08
FNN-24	60.98

Table 2: Training time for 25 TCNs.

Network	Training Time (s)
TCN-0	205.78
TCN-1	205.50
TCN-2	209.29
TCN-3	210.67
TCN-4	192.26
TCN-5	162.44
TCN-6	210.34
TCN-7	179.12
TCN-8	223.55
TCN-9	225.59
TCN-10	203.52
TCN-11	198.07
TCN-12	193.12
TCN-13	238.68
TCN-14	249.54
TCN-15	253.31
TCN-16	249.40
TCN-17	264.71
TCN-18	218.86
TCN-19	269.79
TCN-20	282.67
TCN-21	234.37
TCN-22	298.57
TCN-23	257.24
TCN-24	308.71

EXAMENSARBETE Usage pattern recognition for efficient pre-caching
STUDENTER Otto Sörnäs, Erik Gralén
HANDLEDARE Rasmus Ros (LTH)
EXAMINATOR Emelie Engström (LTH)

Förhämtning av appar i molnmiljö

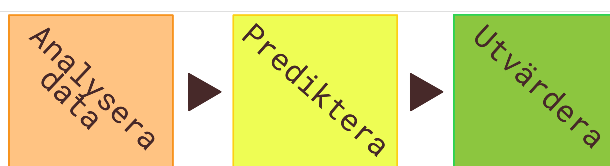
POPULÄRVETENSKAPLIG SAMMANFATTNING **Otto Sörnäs, Erik Gralén**

När en app använder sig av många komplicerade beräkningar kan den från en användares håll upplevas som långsam och trög. Skulle det därför vara möjligt att använda sig av neurala nätverk, en typ av maskininlärning, för att förutspå när en användare tänker öppna en app, och utföra beräkningarna i förväg?

Neurala nätverk, om man applicerar dem på tidsserier, lär sig av historisk data för att gissa sig till hur framtiden kommer att se ut. Inom börshandel, till exempel, så motsvarar detta att man låter ett neuralt nätverk kolla på hur en aktiekurs förändrats under tidsperioden oktober-april. Nätverket lär sig därefter vad det är för underliggande samband i datan som påverkar aktien. Denna kunskap kan nätverket sedan använda för att förutspå hur aktien kommer att utvecklas i mars månad, som den ännu inte har sett.

Denna typ av problem, som kallas för tidsserieprediktion, kan appliceras på många olika områden. I så gott som alla dagens datorer, till exempel, nyttjas en teknik där man i förebyggande syfte förbereder en app en stund innan en användare faktiskt försöker öppna den. Detta görs för att minska den upplevda uppstartstiden. Denna teknik är inbyggd i såväl Windows som Mac och smartphones. Det som är beslutsgrundande för *när* en specifik app ska öppnas är alltsom oftast någon form av maskininlärning. Detta examensarbete har tillämpat två olika neurala nätverk på just detta problem; att identifiera och lära sig ett användarmönster för att kunna öppna en app åt en användare, innan användaren själv försöker öppna appen. De två nätverken har därefter utvärderats, och sedan jämförts mot varandra.

Alla typer av tidsserieprediktion börjar med ett moment som kallas tidsserieanalys. Detta går i det stora hela ut på att man manuellt studerar datan och försöker hitta underliggande mönster som kanske kan hjälpa det neurala nätverket att bli bättre på att prediktera. Tidsbaserade trender i datan är ofta något man försöker hitta. Typiskt för användarmönster i exempelvis kontorsmiljö är att det är högt tryck på måndagsmorgonen, lågt tryck på fredagseftermiddagar, och ingen användning alls på helger.



Resultatet i det här examensarbetet visar att de båda olika neurala nätverken är fullgoda alternativ när det gäller att förutspå när en användare kommer att öppna en app. Det ena nätverket hade lite sämre exakthet när det gällde prediktering, men den var mycket lättare att träna och tog upp mindre utrymme i minnet. Nätverk nummer två hade istället en högre exakthet, men tog mycket längre tid att träna och tog upp mer plats.