Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Ask Moe Løite

Binary Classification of Credit Card Users with Logistic Regression, Gradient Boosted Decision Trees and Deep Learning

Master's thesis in Applied Physics and Mathematics
Supervisor: John Sølve Tyssedal
June 2023

Norwegian University of
Science and Technology

Ask Moe Løite

Binary Classification of Credit Card Users with Logistic Regression, Gradient Boosted Decision Trees and Deep Learning

Master's thesis in Applied Physics and Mathematics
Supervisor: John Sølve Tyssedal
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Preface

This master thesis was written during the spring of 2023 and completes my Master of Science in Applied Physics and Mathematics at the Norwegian University of Science and Technology (NTNU). The thesis was written as a part of my academic specialization within the field of statistics under the guidance of my supervisor, John Sølve Tyssedal. The data for this thesis were provided by Sparebank1, with Christian Meland as a company supervisor. A basic understanding of statistics, statistical modeling, along with familiarity with banking and credit card terminology is expected from the reader.

I would like to thank my supervisor John Sølve Tyssedal, I am grateful for the help and support provided. Additionally, I would like to thank SpareBank1 for giving me the opportunity to write this thesis.

Abstract

This thesis aims to examine passive credit card users within SpareBank1 and focuses on constructing and refining models capable of predicting which passive customers that have the potential to become active again. The task at hand involves binary classification, where the response is categorized as either "true" or "false". The data used for this analysis is provided by SpareBank1 and encompasses data spanning from May 2020 to February 2023.

The predicting models used in this thesis are logistic regression, gradient boosted decision trees and deep learning. These models were initially built with default values for the hyperparameters, and then with optimized values for the hyperparameters found by Bayesian optimization. The predictive models were evaluated by various metrics, including the area under the ROC curve, Matthews Correlation Coefficient, Balanced Accuracy, and the Brier score. Furthermore, the importance of features were investigated using both the predictive model libraries and SHAP values both before and after the hyperparameter tuning process.

Gradient boosted decision trees outperformed both logistic regression and deep learning. Bayesian optimization increased the predicting performance for gradient boosted decision trees and deep learning, however it did not increase the predicting performance for logistic regression. The best AUC (0.6606), Brier score (0.0593), Matthews Correlation Coefficient (0.1190), and Balanced Accuracy (0.6217), where obtained using gradient boosted decision trees. Using optimized hyperparameters did not significantly change the calculated feature importance for the predicting models. Using SHAP values, it was found that different features emerged as the most important features across the different predicting models.

Sammendrag

En betydelig del av den vestlige befolkningen eier et kredittkort, men mange av disse personene bruker ikke aktivt kortene sine. Denne oppgaven har som mål å undersøke passive kredittkortbrukere hos SpareBank1, og fokuserer på å konstruere og forbedre modeller som kan forutsi hvilke passive kredittkort brukere som har potensial til å bli aktive kredittkort brukere. Oppgaven involverer binær klassifisering, der responsen er kategorisert som enten "sann" eller "usann". Dataene som brukes i denne analysen er levert av SpareBank1 og omfatter perioden fra mai 2020 til februar 2023.

I denne oppgaven ble prediksjonsmodellene logistisk regresjon, gradient boosted decision trees og dyp læring brukt. Logistisk regresjon ble valgt på grunnlag av at denne modellen har vært en av de mest brukte binære regresjonsmodellene de siste 40 årene. Dette skyldes enkelheten og tolkningsmulighetene, samtidig som den gir gode prediksjonsresultater med minimal bruk av beregningsressurser. Gradient boosted decision trees ble valgt fordi denne typen modell har vist seg å være allsidig og overgå andre prediksjonsmodeller. Dyp læring ble valgt fordi denne modellen har vist seg å være en svært effektiv maskinlæringsmodell de siste årene, med mange vellykkede anvendelser på tvers av ulike områder.

Prediksjonsmodellene ble først bygget med standardverdier for hyperparametere, og deretter med optimaliserte verdier for hyperparameterne. For å finne de optimale verdiene for hyperparameterne ble Bayesiansk optimalisering brukt. Evalueringen av prediksjonsmodellene fokuserte på AUC, MCC, BACC og Brier-score. Videre ble betydningen av ulike variabler undersøkt ved bruk av både prediksjonsmodellbiblioteker og SHAP-verdier. Denne analysen omfattet vurdering av variablenes betydning både før og etter hyperparameterjusteringen.

Gradient boosted decision trees fikk bedre resultater enn både logistisk regresjon og dyp læring. Bayesiansk optimalisering forbedret prediksjonsytelsen for gradient boosted decision trees og dyp læring, men ikke for logistisk regresjon. De beste verdiene for AUC (0.6606), Brier-score (0.0593), MCC (0.1190) og BACC (0.6217) ble oppnådd med gradient boosted decision trees. Bruk av optimaliserte hyperparametere endret ikke signifikant beregnet betydning av variablene for prediksjonsmodellene. Ved bruk av SHAP-verdier ble det funnet at de viktigste variablene endret seg fra prediksjonsmodell til prediksjonsmodell.

Table of Contents

List of Figures viii

List of Tables xiii

1 Introduction 1

1.1 Methodological background 1

1.2 Outline 2

2 A Brief Introduction to the Data 3

2.1 The Response 3

3 Theoretical Background 6

3.1 Statistical learning 6

3.1.1 Cross-Validation 7

3.1.2 Gradient Descent 8

3.1.3 Evaluation Metrics for Classification 8

3.1.4 SHAP values 12

3.2 Logistic Regression 14

3.2.1 Generalized Linear Models 14

3.2.2 Parameter estimation 15

3.2.3 L1 and L2 Regularization 17

3.2.4 Scikit-learn 17

3.3 Gradient Boosted Decision Trees 18

3.3.1 Decision Trees 18

3.3.2 Ensemble learning 20

3.3.3 Gradient Boosting 20

3.3.4 Gradient Boosted Decision Trees 22

3.3.5 L1 and L2 Regularization . . . 26
3.3.6 LightGBM . . . 26
3.4 Deep Learning . . . 27
3.4.1 The Artificial Neuron . . . 27
3.4.2 The Neural Network . . . 28
3.4.3 Activation functions . . . 30
3.4.4 Training the Neural Network . . . 32
3.4.5 L1 and L2 Regularization . . . 36
3.4.6 Scikit-learn . . . 37
3.5 Hyperparameter tuning . . . 37
3.5.1 Bayesian optimization . . . 38
3.5.2 Optuna . . . 44

4 Data Preparation and Visualization 45
4.1 The different datasets . . . 45
4.2 Visualization . . . 46
4.3 Pre-Processing . . . 48
4.3.1 The first two datasets . . . 48
4.3.2 The last two datasets . . . 50

5 Methods and Hyperparameters, an Overview 52
5.1 The predicting models . . . 52
5.1.1 Logistic Regression . . . 52
5.1.2 Gradient Boosted Decision Trees . . . 53
5.1.3 Deep Learning . . . 54
5.2 The procedure . . . 55
5.2.1 Bayesian Optimization . . . 56
5.2.2 Threshold Optimization . . . 56
5.2.3 SHAP values . . . 56

6 Analysis and Results 57
6.1 Results with default hyperparameters . . . 57
6.1.1 Investigating threshold importance . . . 58
6.1.2 Feature Importance . . . 60
6.2 Logistic Regression Optimization . . . 61
6.2.1 First Results . . . 62

6.2.2 Second Results . . . 65
6.2.3 Threshold . . . 68
6.3 Gradient Boosted Decision Trees Optimization . . . 68
6.3.1 First Results . . . 69
6.3.2 Second Results . . . 71
6.3.3 Threshold . . . 74
6.4 Deep Learning Optimization . . . 75
6.4.1 First Results . . . 75
6.4.2 Second Results . . . 78
6.4.3 Threshold . . . 81
6.5 Comparing the Tuned Models . . . 81
6.5.1 The tuning process . . . 82
6.5.2 Investigating threshold importance . . . 83
6.5.3 Feature importance . . . 85
6.6 Feature Importance through SHAP values . . . 86
6.6.1 Default predicting models . . . 87
6.6.2 Tuned predicting models . . . 89

7 Discussion **91**

8 Conclusion **94**
8.1 Future Work . . . 94

A Variables in the different datasets with explanation **99**
A.1 The fundamental dataset . . . 99
A.2 The appliance dataset . . . 99
A.3 The historical credit card dataset . . . 100
A.4 The historical transactions dataset . . . 101

B Correlation plot **103**

C Results Logistic Regression **104**
C.1 Code Printout . . . 104
C.2 Optimization Plots First Optimization . . . 105
C.3 Optimization Plots Second Optimization . . . 106

D Results Gradient Boosted Decision Trees **108**
D.1 Code Printout . . . 108

D.2 Optimization Plots First Optimization 109
D.3 Optimization Plots Second Optimization 113

E Results Deep Learning 115
E.1 Code Printout 115
E.2 Optimization Plots First Optimization 116
E.3 Optimization Plots Second Optimization 118

F Other outprints and plots 120
F.1 Results 120
F.2 Threshold plots on default models 121
F.3 Threshold plots on tuned models 122
F.4 Printouts from the SHAP values 124
F.4.1 Default Predicting Models 124
F.4.2 Tuned Predicting Models 124

G The Code 126
G.1 Packages 126
G.2 Main Code 126
G.2.1 New Models 129
G.3 Logistic Regression Tuning 130
G.4 Gradient Boosted Decision Trees Tuning 132
G.5 Deep Learning Tuning 133

List of Figures

2.1 Number of observations in each month in the original dataset. 4
2.2 The fraction of observations that becomes active in each month. 5

3.1 An illustration that demonstrates the process of performing 5-fold cross-validation 7
3.2 An illustration showing the confusion matrix 9
3.3 A visual representation that demonstrates the plotting of several ROC curves . . . 12
3.4 A visual representation that demonstrates how a decision tree can perform binary classification 18
3.5 A visual representation that shows how every step in boosting can be compared to hitting a golf ball, gradually moving it closer to the target 21
3.6 A visual representation that shows the main idea behind gradient boosted decision trees. 23
3.7 An illustration provided to clarify how an objective is used in tree boosting 25
3.8 A comparison between a biological neuron and an artificial neuron 27
3.9 An illustration of a neural network 28
3.10 An illustration of a multi layered neural network based on mathematical equations 30
3.11 Different sigmoid functions from different parameters 30
3.12 The derivative of the sigmoid function and the tanh function 31
3.13 An illustration showing the difference between random search and grid search. . . 38
3.14 An illustration showing examples of acquisition functions and their settings 44

4.1 Correlations between different variables within the merged dataset prior to preprocessing 46
4.2 Density plots of selected explanatory variables 47
4.3 The figure shows three box plots on how much money is spent in different categories 48
4.4 Figures showing the importance of keeping information about missing values . . . 50
4.5 An illustration made to show how the historical datasets has been changed to later be merged with the other datasets 50

6.1 A plot illustrating the impact of various threshold values on the Matthews Correlation Coefficient (MCC) of different predictive models with default hyperparameters 59
6.2 A plot illustrating the impact of various threshold values on the Balanced Accuracy (BACC) of different predictive models with default hyperparameters 59
6.3 A plot illustrating the impact of various threshold values on the Accuracy of different predictive models with default hyperparameters 60
6.4 Feature importance for logistic regression with default hyperparameters 60
6.5 Feature importance for gradient boosted decision trees with default hyperparameters and threshold 61
6.6 Optimization history plot for the first Bayesian optimization with logistic regression 63
6.7 Hyperparameter importance plot for the first logistic regression optimization . . . 63
6.8 Slice plot showing the impact the hyperparameter "solver" has on the first logistic regression optimization 64
6.9 Slice plot showing the impact the hyperparameter "C" has on the first logistic regression optimization 64
6.10 Optimization history plot for the second Bayesian optimization with logistic regression 65
6.11 Hyperparameter importance plot for the second logistic regression optimization . . 66
6.12 Slice plot showing the impact the hyperparameter "C" has on the second logistic regression optimization 66
6.13 Contour plot showing how different combinations of "C" and "max_iter" changes the objective value for logistic regression 67
6.14 A plot showing how different values of the threshold affect an objective value for logistic regression 68
6.15 Optimization history plot for the first Bayesian optimization with gradient boosted decision trees 69
6.16 Hyperparameter importance plot for the first gradient boosted decision trees optimization 70
6.17 Slice plot showing the impact the hyperparameter "learning_rate" has on the first gradient boosted decision trees optimization 70
6.18 Slice plot showing the impact the hyperparameter "min_split_gain" has on the first gradient boosted decision trees optimization 71
6.19 Optimization history plot for the second Bayesian optimization with gradient boosted decision trees 72
6.20 Hyperparameter importance plot for the second gradient boosted decision trees optimization 73
6.21 Slice plot showing the impact the hyperparameter "learning_rate" has on the second gradient boosted decision trees optimization 73
6.22 Contour plot showing how different combinations of "learning_rate" and "max_depth" changes the objective value for gradient boosted decision trees 74
6.23 A plot showing how different values of the threshold affect an objective value for gradient boosted decision trees 74
6.24 Optimization history plot for the first Bayesian optimization with deep learning . . 76

6.25 Hyperparameter importance plot for the first deep learning optimization 76
6.26 Slice plot showing the impact the hyperparameter "num_layers" has on the first deep learning optimization 77
6.27 Slice plot showing the impact the hyperparameter "activation" has on the first deep learning optimization 77
6.28 Optimization history plot for the second Bayesian optimization with deep learning 79
6.29 Hyperparameter importance plot for the second deep learning optimization 79
6.30 Slice plot showing the impact the hyperparameter "num_layers" has on the second deep learning optimization 80
6.31 Contour plot showing how different combinations of "num_layers" and "max_iter" changes the objective value for gradient boosted decision trees 80
6.32 A plot showing how different values of the threshold affect an objective value for deep learning 81
6.33 A plot illustrating the impact of various threshold values on the Matthews Correlation Coefficient (MCC) of different predictive models with optimal hyperparameters 84
6.34 A plot illustrating the impact of various threshold values on the Balanced Accuracy (BACC) of different predictive models with optimal hyperparameters 84
6.35 A plot illustrating the impact of various threshold values on the Accuracy of different predictive models with optimal hyperparameters 85
6.36 Feature importance for logistic regression with optimal hyperparameters 85
6.37 Feature importance for gradient boosted decision trees with optimal hyperparameters 86
6.38 A figure showcasing the SHAP values for the default logistic regression model . . . 88
6.39 A figure showcasing the SHAP values for the default gradient boosted decision trees model 88
6.40 A figure showcasing the SHAP values for the default deep learning model 89
6.41 A figure showcasing the SHAP values for the tuned logistic regression model 90

B.1 Correlations in the merged dataset between the fundamental dataset and the appliance dataset after pre-processing 103

C.1 Printout of the best hyperparameter values and the execution time for the first optimization with Logistic Regression 104
C.2 Printout of the best hyperparameter values and the execution time for the second optimization with Logistic Regression 104
C.3 Slice plot showing the impact the hyperparameter "penalty" has on the first Logistic Regression optimization 105
C.4 Slice plot showing the impact the hyperparameter "max_iter" has on the first Logistic Regression optimization 105
C.5 Slice plot showing the impact the hyperparameter "tol" has on the first Logistic Regression optimization 106
C.6 Slice plot showing the impact the hyperparameter "max_iter" has on the second Logistic Regression optimization 106

C.7 Slice plot showing the impact the hyperparameter ”tol” has on the second Logistic Regression optimization 107

D.1 Printout of the best hyperparameter values and the execution time for the first optimization with Gradient Boosted Decision Trees 108

D.2 Printout of the best hyperparameter values and the execution time for the second optimization with Gradient Boosted Decision Trees 108

D.3 Slice plot showing the impact the hyperparameter ”colsample_bytree” has on the first Gradient Boosted Decision Trees optimization 109

D.4 Slice plot showing the impact the hyperparameter ”max_depth” has on the first Gradient Boosted Decision Trees optimization 109

D.5 Slice plot showing the impact the hyperparameter ”reg_alpha” has on the first Gradient Boosted Decision Trees optimization 110

D.6 Slice plot showing the impact the hyperparameter ”subsample” has on the first Gradient Boosted Decision Trees optimization 110

D.7 Slice plot showing the impact the hyperparameter ”num_leaves” has on the first Gradient Boosted Decision Trees optimization 111

D.8 Slice plot showing the impact the hyperparameter ”reg_lambda” has on the first Gradient Boosted Decision Trees optimization 111

D.9 Slice plot showing the impact the hyperparameter ”n_estimators” has on the first Gradient Boosted Decision Trees optimization 112

D.10 Slice plot showing the impact the hyperparameter ”colsample_bytree” has on the first Gradient Boosted Decision Trees optimization 112

D.11 Slice plot showing the impact the hyperparameter ”min_child_samples” has on the first Gradient Boosted Decision Trees optimization 113

D.12 Slice plot showing the impact the hyperparameter ”max_depth” has on the second Gradient Boosted Decision Trees optimization 113

D.13 Slice plot showing the impact the hyperparameter ”n_estimators” has on the second Gradient Boosted Decision Trees optimization 114

E.1 Printout of the best hyperparameter values and the execution time for the first optimization with Deep Learning 115

E.2 Printout of the best hyperparameter values and the execution time for the second optimization with Deep Learning 115

E.3 Slice plot showing the impact the hyperparameter ”alpha” has on the first Deep Learning optimization 116

E.4 Slice plot showing the impact the hyperparameter ”learning_rate_init” has on the first Deep Learning optimization 116

E.5 Slice plot showing the impact the hyperparameter ”max_iter” has on the first Deep Learning optimization 117

E.6 Slice plot showing the impact the hyperparameter ”neurons_per_layer” has on the first Deep Learning optimization 117

E.7 Slice plot showing the impact the hyperparameter ”tol” has on the first Deep Learning optimization 118

E.8 Slice plot showing the impact the hyperparameter "max_iter" has on the second Deep Learning optimization . . . 118

E.9 Slice plot showing the impact the hyperparameter "neurons_per_layer" has on the second Deep Learning optimization . . . 119

F.1 Results from classification metrics on all models. To the left default hyperparameters and threshold are used. To the right tuned hyperparameters and threshold are used. 120

F.2 Confusion matrices for all the models. To the left default hyperparameters and threshold are used. To the right tuned hyperparameters and threshold are used. . . 121

F.3 A plot illustrating the impact of various threshold values on the Sensitivity of different predictive models with default hyperparameters . . . 121

F.4 A plot illustrating the impact of various threshold values on the Specificity of different predictive models with default hyperparameters . . . 122

F.5 A plot illustrating the impact of various threshold values on the Sensitivity of different predictive models with optimal hyperparameters . . . 123

F.6 A plot illustrating the impact of various threshold values on the Specificity of different predictive models with optimal hyperparameters . . . 123

F.7 A figure showcasing the SHAP values for the tuned deep learning model . . . 124

F.8 Runtime for getting the SHAP values from the default Logistic Regression model . 124

F.9 Runtime for getting the SHAP values from the default Gradient Boosted Decision Trees model . . . 124

F.10 Runtime for getting the SHAP values from the default Deep Learning model . . . 124

F.11 Runtime for getting the SHAP values from the tuned Logistic Regression model . . 124

F.12 Anticipated runtime for getting the SHAP values from the tuned Gradient Boosted Decision Trees model . . . 125

F.13 Runtime for getting the SHAP values from the tuned Deep Learning model . . . 125

List of Tables

2.1 The average amount of people that becomes active or remain passive in each month. 3

3.1 An example of how Shapley values are calculated 14

4.1 A summary of all the different datasets. 45

5.1 The hyperparameters selected for tuning logistic regression, along with their default values, data type, and domain. 53

5.2 The hyperparameters selected for tuning gradient boosted decision trees, along with their default values, data type, and domain. 54

5.3 The hyperparameters selected for tuning deep learning, along with their default values, data type, and domain. 55

6.1 Confusion matrix from logistic regression. The training of the model was performed on the training set using default hyperparameters, and the model was evaluated on the test set using a default threshold of 0.064. 0 represents remaining passive while 1 represents becoming active. 57

6.2 Confusion matrix from gradient boosted decision trees. The training of the model was performed on the training set using default hyperparameters, and the model was evaluated on the test set using a default threshold of 0.064. 0 represents remaining passive while 1 represents becoming active. 58

6.3 Confusion matrix from deep learning. The training of the model was performed on the training set using default hyperparameters, and the model was evaluated on the test set using a default threshold of 0.064. 0 represents remaining passive while 1 represents becoming active. 58

6.4 Results from classification metrics on the test set for all models with default hyperparameters and threshold. 58

6.5 The hyperparameters with their search domain for the first logistic regression hyperparameter tuning. 62

6.6 The hyperparameters with their optimal values for the first logistic regression hyperparameter tuning. 62

6.7 The hyperparameters with their respective search domain for the second logistic regression hyperparameter tuning. 65

6.8 The hyperparameters with their optimal values for the second logistic regression hyperparameter tuning. 65

6.9 The hyperparameters with their respective search domain for the first gradient boosted decision trees hyperparameter tuning. 68

6.10 The hyperparameters with their optimal values for the first gradient boosted decision trees hyperparameter tuning. 69

6.11 The hyperparameters with their respective search domain for the second gradient boosted decision trees hyperparameter tuning. 71

6.12 The hyperparameters with their optimal values for the second gradient boosted decision trees hyperparameter tuning. 72

6.13 The hyperparameters with their respective search domain for the first deep learning hyperparameter tuning. 75

6.14 The hyperparameters with their optimal values for the first deep learning hyperparameter tuning. 75

6.15 The hyperparameters with their respective search domain for the second deep learning hyperparameter tuning. 78

6.16 The hyperparameters with their optimal values for the second deep learning hyperparameter tuning. 78

6.17 Confusion matrix from logistic regression. The training of the model was performed on the training set using optimal hyperparameters, and the model was evaluated on the test set using a optimal threshold of 0.056. 0 represents remaining passive while 1 represents becoming active. 81

6.18 Confusion matrix from gradient boosted decision trees. The training of the model was performed on the training set using optimal hyperparameters, and the model was evaluated on the test set using a optimal threshold of 0.060. 0 represents remaining passive while 1 represents becoming active. 82

6.19 Confusion matrix from deep learning. The training of the model was performed on the training set using optimal hyperparameters, and the model was evaluated on the test set using a optimal threshold of 0.062. 0 represents remaining passive while 1 represents becoming active. 82

6.20 Results from classification metrics on the test set for all predicting models with optimal hyperparameters and threshold. 82

6.21 Information regarding the tuning process for each predicting model. The best objective value and the runtime is displayed for both the first and the second Bayesian optimization 83

6.22 The difference in the results from classification metrics on all models before and after tuning. 83

6.23 The execution time for getting the SHAP values for the different predictive models, both with and without optimized hyperparameters 87

A.1 Explanation of all the variables in the fundamental dataset 99

A.2 Explanation of all the variables in the appliance dataset 99

A.3 Explanation of all the variables in the historical credit card dataset 100

A.4 Explanation of all the variables in the historical transactions dataset 101

Chapter 1

Introduction

The use of credit cards has become an important part of the modern world, with many people relying on them for various transactions. However, the origins of the credit card usage can be traced back to the United States during the 1920s. During this time, individual firms such as oil companies and hotel chains began to issue credit cards to their customers for purchases made at their respective outlets [41].

Despite the introduction of these company-specific credit cards, it wasn't until 1950 that the first universal credit card was founded. This innovation is credited to Diners Club, which allowed customers to use their credit cards at a variety of different establishments beyond those owned by a particular company. The founding of Diners Club is considered a significant milestone in the history of credit cards as it paved the way for the widespread adoption of credit cards by consumers and businesses alike.

Today, credit cards are used by millions of people around the world for purchases of all kinds, ranging from everyday essentials to major investments. With the rise of digital payments and contactless technology, credit cards continue to evolve to meet the changing needs of consumers in the 21st century. According to a report conducted by SIFO [35], in 2019, 87 percent of the adult population in Norway possessed one or more credit cards.

The source of revenue for credit card companies comes from collecting various fees, among which interest charges are the most significant. If a user fails to pay back the balance in full each month, they are charged interest. Late fees and annual fees also contribute to the revenue, but to a lesser extent. Consequently, credit card companies may potentially suffer losses if cardholders do not use their credit cards. Thus, it can be advantageous for companies offering credit cards to anticipate which of the inactive cardholders that will continue to remain inactive.

A cardholder who has not used their card within the last six months is classified as a passive user, while a cardholder who has used their card within the last six months is categorized as an active user.

1.1 Methodological background

Predicting binary outcomes is an important task in many modern world aspects. Advanced modeling techniques have the potential to provide more precise predictions of binary outcomes compared to classical techniques. Logistic regression has for a long time been the foundation of most prediction models with binary outcomes, as it is a reliable method for binary classification. Logistic regression offers several advantages, such as ease of implementation, low computational cost and interpretable results. Moreover, in many cases, the predictive performance of more advanced models is not substantially better than that of logistic regression. In some situations, particularly when

data is scarce, or when there is a need to use minimal computing power, logistic regression can even outperform more sophisticated methods [40].

Over the past decades, an increasing amount of data has been converted into digital form, while computing power has experienced a remarkable increase. As a result, there has been a growing tendency among people to move away from traditional statistical learning and adopt machine learning instead. Additionally, machine learning algorithms have gained significant traction in research, and many cutting-edge algorithms are now accessible to the public for free.

Machine learning is essentially a scientific exploration of statistical models and intricate algorithms that primarily rely on patterns and inference. With the promise of capturing non-linearities and interactions in the data more effectively than classical statistical processes, machine learning has the potential to deliver superior results.

Deep learning models have also received a lot of attention and expectations. These models can learn hierarchical representations of the data, which can capture complex relationships between features. As a result, neural networks have achieved state-of-the-art performance in many classification tasks.

Although machine learning has many benefits, there are also some drawbacks. For example, modern techniques greater flexibility means that for reliable estimation, there may be needed larger sample sizes. Additionally, machine learning algorithms can be impractical due to the need to adjust hyperparameters, their high computational cost, and the fact that the resulting model may be difficult to interpret.

1.2 Outline

In this thesis real world credit card data has been provided by Sparebank1. The task at hand is to predict which passive credit card users that will convert into active users month for month. This task involves binary classification, where a response of 1 denotes a user who becomes active, and 0 denotes one who remains passive. To accomplish this goal, this thesis employs three predicting models; logistic regression, gradient boosted decision trees, and deep learning. Logistic regression is chosen due to it being one of the most used binary classification models for the last 40 years, which is because of its simplicity and interpretability, while also producing good predicting results with a minimum use of computational resources. Gradient boosted decision trees is chosen because it has been shown to be versatile, and outperform other predicting models when presented with much data. Deep learning is chosen as it has proven to be a highly effective machine learning technique in recent years, with numerous successful applications across a diverse range of fields. Furthermore, Bayesian optimization will be employed to optimize the hyperparameters of each prediction model for optimal results.

The structure of this thesis is as follows: Chapter 2 gives a short introduction to the data provided by Sparebank1, and the response variable. Chapter 3 provides the relevant theoretical background for the prediction and evaluation methods used, along with the topic of hyperparameter tuning and Bayesian optimization. Chapter 4 includes investigation and modification of the dataset, this encompasses data visualization, feature engineering, and other pre-processing techniques. Chapter 5 gives an overview of the methods used for training and testing. This includes a description of each of the hyperparameters that is being optimized in the training. Chapter 6 presents the main results. Chapter 7 covers the discussion of results, while Chapter 8 provides concluding remarks.

Chapter 2

A Brief Introduction to the Data

Sparebank1 provided the datasets used in this thesis. Originally four datasets were given, where one of them forms the basis of the predicting task. The three remaining datasets consists of historical credit card usage, historical transactions, along with data gathered when the customer applied for the given credit card. Combining all four datasets demands some pre-processing, however it is the basis dataset that is the fundamental dataset for the prediction. The other three datasets provides additional information.

The fundamental dataset consists of 262773 observations and 12 variables. The variables contain information regarding the customer, such as what sex the customer are, the age of the customer, at what services the customer got their credit card, the first and last time the customers used their credit card, the period the prediction takes place, and so on. Appendix A contains a comprehensive list of all the variables in all the original datasets along with their respective explanations.

Each observation in the dataset represents a single customer at a given date. The dates range from May 2020 all the way to February 2023.

2.1 The Response

Each observation will have the response variable ”AktivEtterPassiv”. This variable is a binary response variable that tells if a customer has become active in a given month. It will be one if a customer becomes active in that one month period and zero if the customer remains passive in that one month period.

This dataset consists of passive credit card users only, so if a customer becomes active in a given month, that customer will then not be included in the dataset for the next month. Customers that do not use their credit card for six months are classified as passive users, and they will then be included in the dataset.

It can be observed from Table 2.1 that only a small fraction of the observations become active each month. This means that the majority of the customers that are passive remains passive.

Table 2.1: The average amount of people that becomes active or remain passive in each month.

Type	Remaining passive(0)	Becoming active(1)	Total
Absolute	246020	16753	262773
Percentage	93.6 %	6.4 %	100 %

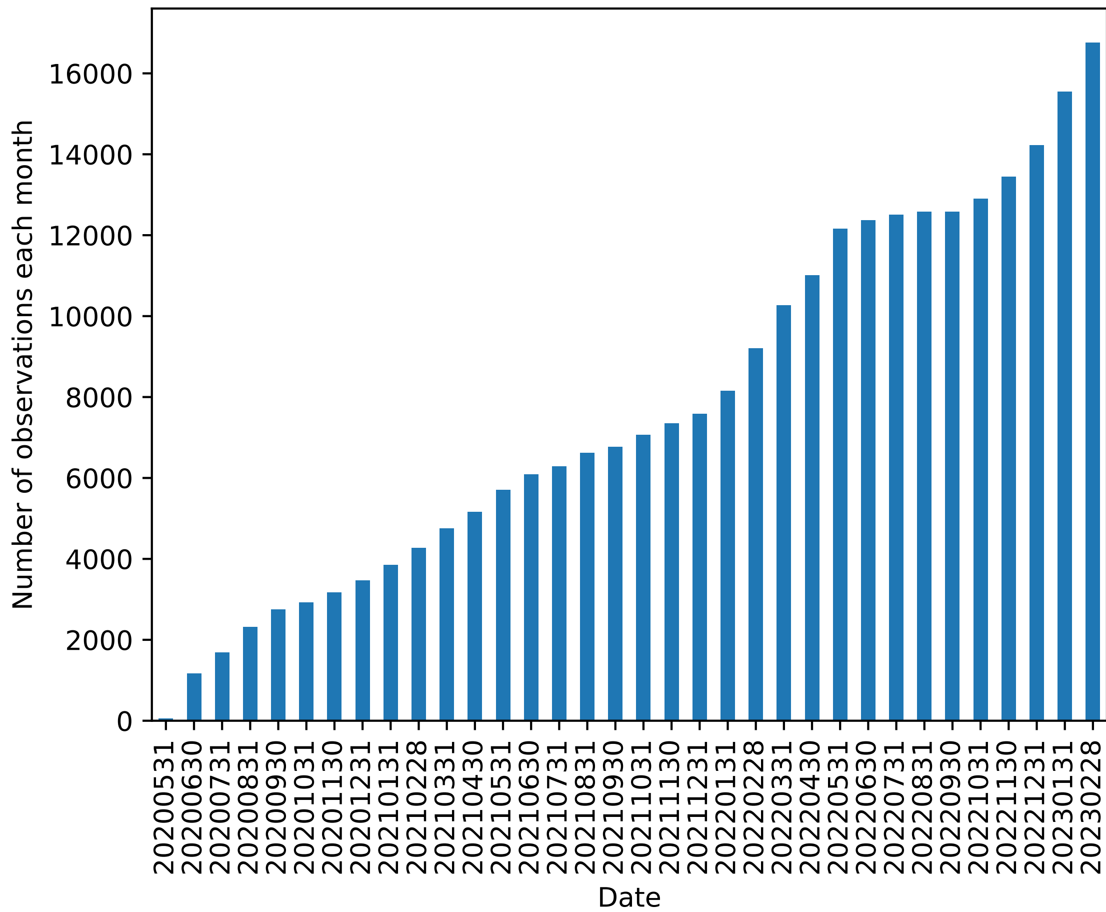

Figure 2.1: A figure showing the number of observations in each month. The dates are on the format yyyymmdd.

Figure 2.1 shows the amount of observations in each month. One can notice that the number of observations only increases each month, with one exeption; from August 2022 to September 2022 there is a small, almost unnoticeable decrease in observations. This decline is believed to come from the fact that August 2022 is the month in the dataset where the most amount of people became active again. These customers would thus exit the dataset.

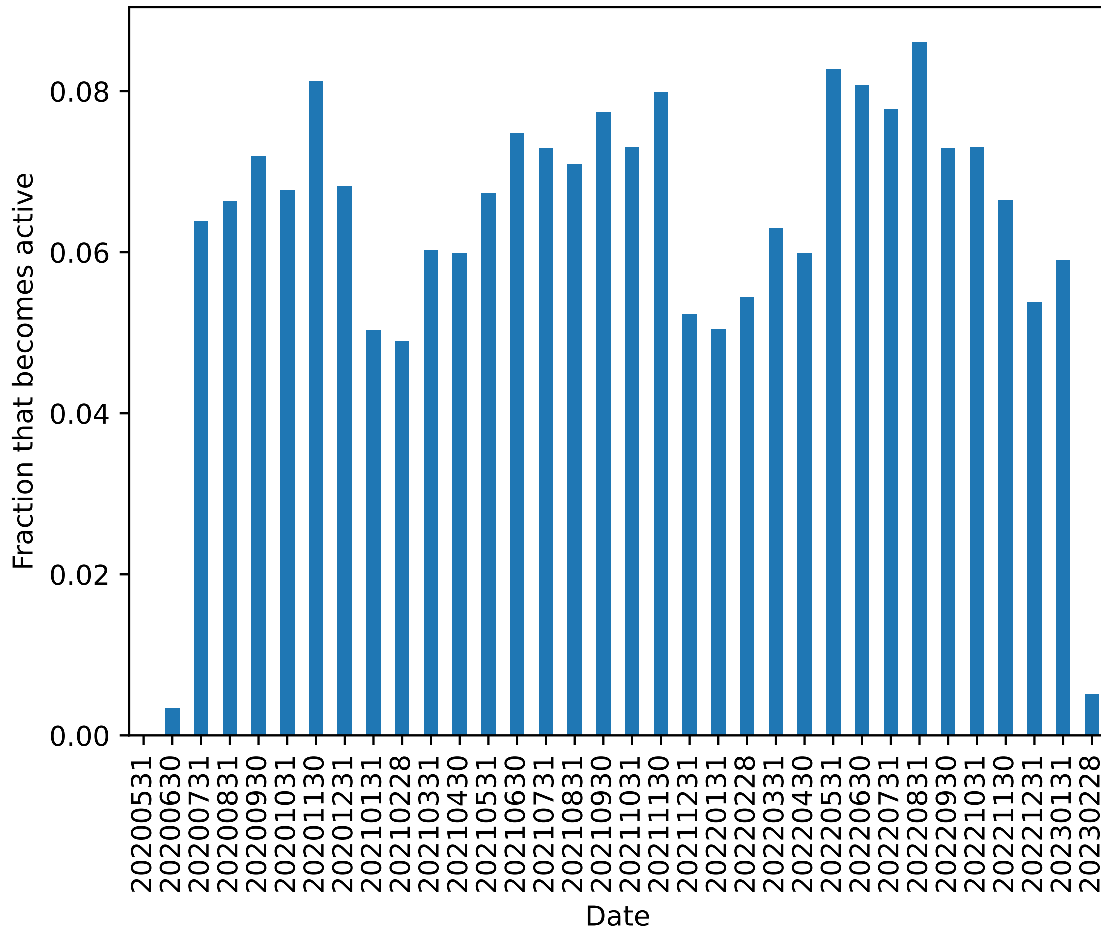

Figure 2.2: A figure showing the fraction of observations that becomes active in each month. The dates are on the format yyyymmdd.

Figure 2.2 shows the fraction of passive customers that becomes active in each month. One can notice that there is sort of a pattern. The fraction of people that becomes active increases in the warmer months, while it decreases in the colder months. One can also notice that the base level of customers that becomes active are more or less constant throughout the prediction period.

Chapter 3

Theoretical Background

This chapter will seek to explain the theoretical foundations of the methods used in this thesis. There will first be introduced some basic terms in statistical learning. Logistic regression, gradient boosted decision trees and deep learning will then be covered. Hyperparameter tuning with Bayesian optimization will be presented at the end.

Some part of this chapter draws upon the theoretical framework of my project thesis [27] written in the fall of 2022. Because certain illustrations and formulas were deemed significant enough to be included yet again, this chapter will contain some repetition in the form of figures and formulas.

3.1 Statistical learning

The primary objective of statistical learning theory is to establish a framework for investigating the problem of inference, which encompasses gaining knowledge, making predictions, making decisions, or constructing models from a given dataset [29]. Statistical learning is generally categorized into two broad groups: unsupervised and supervised learning. Unsupervised learning concerns itself with comprehending data points that consist of explanatory variables exclusively, without a linked label. The purpose of unsupervised learning is to attempt to describe how the data are related and structured, and to identify meaningful patterns. In contrast, supervised learning tries to explain how an output variable is correlated with its associated explanatory variables, establish a model, and make predictions about the outcome. This thesis will focus on supervised learning.

In supervised learning the primary objective is to identify a function F that can predict a response variable y using explanatory variables $\boldsymbol{x}$, this can be represented as

$$y = F(\boldsymbol{x}). \tag{3.1}$$

The explanatory variables $\boldsymbol{x}$ are typically a vector of multiple components.

The primary focus of this thesis is to explore various methods for predicting real-world data. In many real-world scenarios, it is often impossible to make perfect predictions of an outcome from a set of variables, this is due to the presence of noise and other sources of variability. Therefore, when performing predictions using such data, there will almost always be some degree of error. One of the key objectives of this thesis is to compare and evaluate different predictive models to determine the one that yields the best results. To achieve this, it is necessary to estimate the error associated with each model accurately.

One commonly used method for estimating the error associated with a predictive model is to randomly split the available data into two distinct sets: a training set and a test set. The training

set is then used to train the predictive model F, while the test set is held out for evaluating the model's performance.

Once the model F has been trained on the training set, it can be used to predict the response variable y for the explanatory variables $\boldsymbol{x}$ in the test set. The predicted response $\hat{y}$ can then be compared to the true response y using an error function to obtain an estimate of the model's accuracy on the test data.

3.1.1 Cross-Validation

In situations where the data is rich, it can be beneficial to split the dataset into three parts: a training set, a validation set, and a test set. After fitting a model on the training set, the validation set can be used as an evaluation to determine if the model needs to be modified, or if it meets the desired results. Finally, the test set is used for the final evaluation of the model's performance.

The thought of evaluation can be taken further to something called cross-validation. The two primary forms of cross-validation are K-fold cross-validation and leave-one-out cross-validation, where leave-one-out cross-validation is a special case of K-fold cross-validation.

In K-fold cross-validation, the training data is split into K equally sized groups. One of the groups is used for evaluation, while the remaining $K-1$ groups are used for training a model. This process is repeated with a different group as the evaluation group until all K groups have been used for evaluation. The prediction error from each evaluation is then averaged to obtain an estimate of the model's performance. This procedure is illustrated in Figure 3.1.

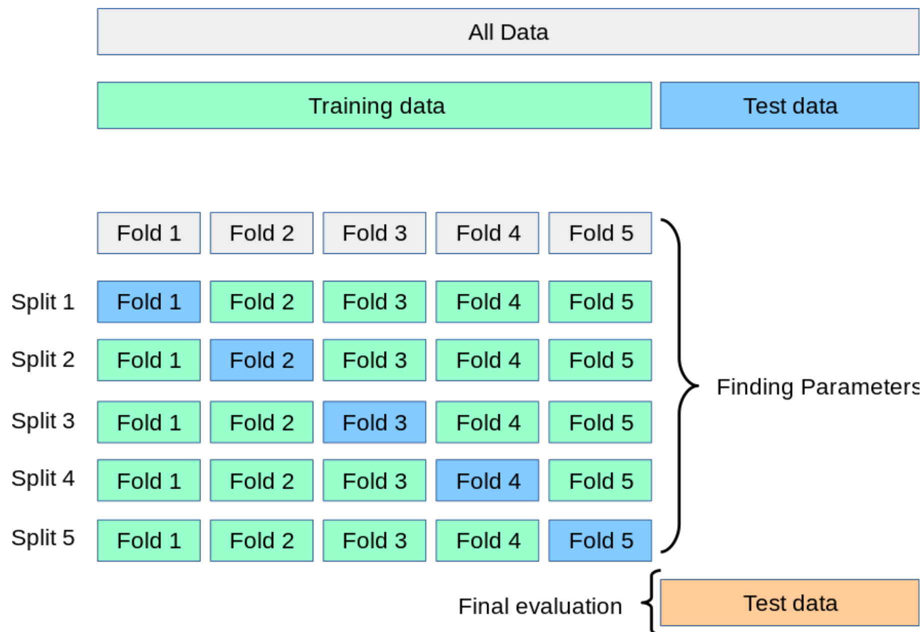

Figure 3.1: An illustration that demonstrates the process of performing 5-fold cross-validation. Taken from [13].

The choice of the optimal number of folds K in K-fold cross-validation is a trade-off between prediction performance and computational cost. Increasing K typically leads to better predictions but also increases the computational cost. A common value for K is 5, which strikes a good balance between prediction performance and computational efficiency.

However, if the dataset is small or computational cost is not a concern, leave-one-out cross-validation can be used, where $K = N$, with N being the number of observations in the training set. In leave-one-out cross-validation, each observation is used as the validation set exactly once, which can reduce bias in the estimate of the prediction error. However, it can have high variance

and be sensitive to outliers.

The main advantage of cross-validation is that it allows for a more reliable estimate of a model's performance on new data than simple train-test splitting. Cross-validation also enables the tuning of model hyperparameters.

3.1.2 Gradient Descent

Gradient descent is a widely used optimization algorithm that helps find the local minimum or maximum of a given function. In the context of machine learning, this algorithm is commonly used to minimize a cost or loss function by iteratively updating a set of parameters. At each iteration, the algorithm computes the gradient of the loss function with respect to the parameters and moves in the direction of the steepest descent until a local minimum is reached.

The basics of the gradient decent algorithm follow these steps. At the start a set of parameters $\boldsymbol{p}$ are often given or randomly initiated. The gradient descent algorithm then calculates the next set of parameters using the current parameters, the gradient of the current parameters, and a stepsize hyperparameter ρ:

$$\boldsymbol{p}_{n+1} = \boldsymbol{p}_n - \rho \nabla F(\boldsymbol{p}_n) \tag{3.2}$$

The hyperparameter ρ in the gradient descent algorithm is commonly known as the step size. This hyperparameter often controls the learning rate of more advanced machine learning algorithms. Choosing an appropriate learning rate is often very important for achieving good performance of the model.

3.1.3 Evaluation Metrics for Classification

In supervised learning, the ultimate goal is to build a model that accurately can predict on independent test data. However, in real-world scenarios there will almost always be some kind of error present. Therefore, it is crucial to evaluate the performance of the model. By measuring the model's performance on an independent test set, one can determine the accuracy of the model's predictions and make informed decisions about its suitability for a given task. By using appropriate evaluation metrics, one can build models that are optimized for the specific task at hand and that provide reliable, accurate predictions for a wide range of inputs.

In binary classification, the task is to assign each data point to one of two categories, often labeled as positive (1) or negative (0). Many binary classification models work by predicting a probability $p \in [0, 1]$, which represents the estimated likelihood that a given data point belongs to the positive class. To make a final classification decision, a threshold is chosen such that all data points with a predicted probability greater than the threshold are classified as positive, while those with a probability less than or equal to the threshold are classified as negative.

There are two types of binary classification metrics that are used to evaluate the performance of such models. The first type of metric is designed to be used with binary data where a threshold is chosen. These metrics provide information about the model's ability to correctly classify positive and negative instances based on a specific threshold value.

The second type of binary classification metrics is designed to evaluate the model's performance irrespective of the specific classification threshold chosen. These metrics are based on the ranking of the predicted probabilities p, rather than the final classification of each observation.

Evaluation with threshold

Evaluating a model when a threshold is chosen and the predictions are binary requires the use of metrics that are all based on the confusion matrix [10]. The confusion matrix is a 2×2 matrix

commonly used to show the count of correct and incorrect predictions. It includes four different outcomes: true positive (TP), true negative (TN), false positive (FP), and false negative (FN). TP represents a positive prediction correctly classified, TN represents a negative prediction correctly classified, FP represents a positive prediction incorrectly classified, and FN represents a negative prediction incorrectly classified. Figure 3.2 displays the confusion matrix in an intuitive way. By using the confusion matrix, many different evaluation metrics can be calculated.

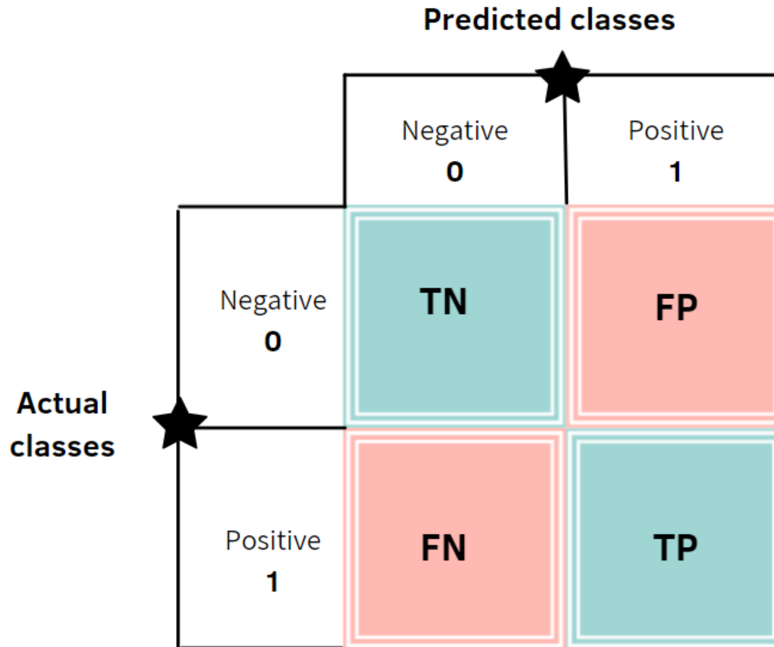

Figure 3.2: An illustration of the confusion matrix. Taken from [22].

Accuracy

A commonly used metric to evaluate the performance of a binary classification model is accuracy, which calculates the proportion of correct predictions among all predictions made. Accuracy is defined as

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TN} + \text{TP} + \text{FP} + \text{FN}}. \tag{3.3}$$

Accuracy is a popular go-to metric as it provides a simple and intuitive measure of the overall performance of the model, as it shows just how accurate the model is. However, accuracy can be misleading if the data is imbalanced. Then, it is easy to get a high accuracy score by simply classifying all observations as belonging to the majority class.

Sensitivity and Specificity

Sensitivity or True Positive Rate (TPR) measures the proportion of actual positive observations that have been classified as positive by the model. Sensitivity is defined as

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{3.4}$$

Sensitivity is a crucial metric in situations where missing a positive response could have severe consequences, such as identifying a dangerous disease.

Specificity or True Negative Rate (TNR) on the other hand measures the proportion of actual negative observations that have been classified as negative by the model. Specificity is defined as

$$\text{TNR} = \frac{\text{TN}}{\text{TN + FP}}. \tag{3.5}$$

In situations where identifying a negative response carries high stakes, Specificity plays a critical role as a metric.

While both Specificity and Sensitivity can be important metrics to use, relying solely on either one of them can lead to incomplete evaluation of the model's performance, as these metrics tell only a part of the whole prediction.

Type I and II Error

Type I error, or false positive rate (FPR) measures the proportion of actual negative observations that have been classified as positive by the model, and is defined as

$$\text{FPR} = \frac{\text{FP}}{\text{FP + TN}}. \tag{3.6}$$

Type II error, or false negative rate (FNR) measures the proportion of actual positive observations that have been classified as negative by the model, and is defined as

$$\text{FNR} = \frac{\text{FN}}{\text{FN + TP}}. \tag{3.7}$$

Both of these metrics are rarely used alone, but they can be important in cases where monitoring errors is crucial.

Each of the classification metrics mentioned above has a range of values between 0 and 1. Generally, a higher value for accuracy, specificity, and sensitivity indicates a better predictive model, while a higher value for Type I and II error typically indicates a worse predictive model.

Balanced Accuracy

Balanced Accuracy (BACC) is a metric that uses the mean value of the sensitivity metric and the specificity metric. Balanced Accuracy is defined as

$$\text{BACC} = \frac{1}{2}(\text{Specificity} + \text{Sensitivity}) = \frac{1}{2}\left(\frac{\text{TN}}{\text{TN + FP}} + \frac{\text{TP}}{\text{TP + FN}}\right) \tag{3.8}$$

The Balanced Accuracy falls within the range of $[0, 1]$, and a greater value indicates better predictive capability. Balanced Accuracy is particularly helpful for imbalanced data sets, as a higher score suggests strong performance in all areas of the confusion matrix.

Matthews Correlation Coefficient

The Matthews Correlation Coefficient (MCC) is a metric that quantifies the correlation between predicted classes and actual data, and it can be expressed as

$$\text{MCC} = \frac{\text{TP} \cdot \text{TN - FP} \cdot \text{FN}}{\sqrt{\text{(TN + FP)(TP + FN)(TN + FP)(TN + FN)}}}. \tag{3.9}$$

The Matthews Correlation Coefficient falls within the range of $[-1, 1]$, where 1 is a perfect prediction and -1 is the opposite. A value of 0 indicates a random classifier.

The Matthews Correlation Coefficient has several advantages [10]. It performs well with imbalanced data and is insensitive to the threshold value. A high MCC indicates excellent performance across all the outcomes of the confusion matrix.

Evaluation without threshold

When the threshold is not set, there are plenty of metrics available to assess the model's performance. The objective is to evaluate the probabilistic predictions p compared to the actual values y.

Brier Score

The Brier Score (BS) is a measure of the accuracy of probabilistic predictions, which calculates the mean squared difference between predicted probabilities and the actual outcomes of events. The Brier Score is defined as

$$\mathrm{BS} = \frac{1}{N}\sum_{i=1}^{N}(y_i - p_i)^2. \tag{3.10}$$

The Brier score ranges from 0 to 1, with 0 indicating perfect predictions. The Brier score is commonly used to calibrate probabilistic predictions, which involves adjusting the predicted probabilities to better reflect the true probabilities of the events. This can help to improve the overall accuracy of the predictions and reduce the potential for bias or error in the model.

AUC and ROC Curve

The ROC curve is a graphical representation of the relationship between the true positive rate (TPR) and false positive rate (FPR). This involves plotting the TPR and FPR at each threshold value on a chart to gain a understanding of their tradeoff, as showed in Figure 3.3.

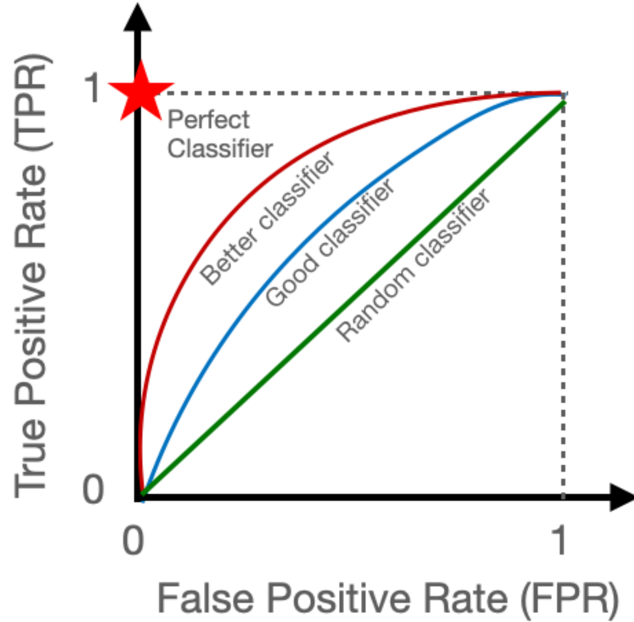

Figure 3.3: A visual representation that demonstrates the plotting of several ROC curves. Taken from [39]

A model's performance can be evaluated by analyzing its TPR and FPR values, with high TPR and low FPR indicating a good model. The ROC AUC score, commonly called AUC, is calculated by measuring the area under the ROC curve, hence providing a numerical representation of the model's performance. This score ranges from 0 to 1, with higher values indicating better model performance, and where a completely random classifier would have an AUC of $\frac{1}{2}$. The AUC is a popular and reliable metric that provides a balanced evaluation of the model's performance for both positive and negative classes.

3.1.4 SHAP values

SHAP (SHapley Additive exPlanations) values are a values used to explain the output of any machine learning model by determining the contribution of each feature to the model's predictions [5]. SHAP values was first proposed by Lundberg and Lee in 2017, and are based on the Shapley values from cooperative game theory introduced by Lloyd Shapley back in 1953.

One single Shapley value can be understood as the average incremental contribution towards the prediction made by a observation of a single feature, taking into account all conceivable combinations of features. In essence, Shapley values are the evaluation of the significance of each feature by measuring its impact on the model's prediction given all the other features.

Consider a predicting model with k features, such that the feature values are given by $\boldsymbol{x} = \big(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k\big)$ [38]. Furthermore, consider how to compute the significance of one particular feature, call this feature I. First, let S be a combination of features such that $S \subseteq Q \setminus \{I\}$, where $Q \setminus \{I\}$ is the set of all the k features except the feature I. Now, to compute the effects of the feature I, let us consider two predictions from the same model: The first one is a prediction with the combination of features S and the feature I, denoted $F(\boldsymbol{x}_{S\cup I})$. The second one is a prediction with just the features S, denoted $F(\boldsymbol{x}_S)$. The two predictions can now be compared through subtraction to see how the much the model gets influenced by the feature I,

$$F(\boldsymbol{x}_{S\cup I}) - F(\boldsymbol{x}_S). \tag{3.11}$$

When considering the impact of excluding a specific feature from the model, it is essential to take into account that the results one get depends upon the other features in the model. This means that to fully understand how much the feature I contributes to the model, one have to be considering all possible combinations of features in $Q \setminus \{I\}$. All these combinations are summed together and weighted like this

$$\sum_{S \subseteq Q \setminus \{I\}} \frac{|S|!(k - |S| - 1)!}{k!}. \tag{3.12}$$

Here $|S|$ denotes the amount of features in S. This weighing acts as a averaging over all possible feature orderings. Now, let the Shapley value for feature I be denoted ϕ_I. This is how much the feature I contributes to the model output. The contribution for feature I is as a result given by

$$\phi_I = \sum_{S \subseteq Q \setminus \{I\}} \frac{|S|!(k - |S| - 1)!}{k!} \Big(F(\boldsymbol{x}_{S \cup I}) - F(\boldsymbol{x}_S) \Big). \tag{3.13}$$

This can be done for all the features in the dataset. The resulting Shapley values represent the degree of influence of each feature on the model output. Positive Shapley values indicate that a feature positively contributes to the model output, while negative Shapley values indicate the opposite.

To get a further understanding of how a Shapley algorithm works, let us consider an example of a model with three features, A, B and C. One can now consider the following outputs,

$$F(x_A) = 5, \quad F(x_B) = 3, \quad F(x_C) = -1,$$

$$F(x_{A \cup B}) = 7, \quad F(x_{A \cup C}) = 3, \quad F(x_{B \cup C}) = 0,$$

and,

$$F(x_{A \cup B \cup C}) = 8.$$

Because there are three features, there are 3! ways of arranging these features if one were to put all three of them after one another, this is displayed in Table 3.1. For each of these arrangements one can calculate how much each feature contributes to the model prediction. Let us consider the first row in Table 3.1, $F(x_A) = 5$, so feature A alone contributes 5. Including feature B in a coalition where A is already there, gives $F(x_{A \cup B}) = 7$, now one can see how much feature B adds to the predicted value: $F(x_{A \cup B}) - F(x_A) = 7 - 5 = 2$. To include yet another feature, feature C, yields an effect of 1 since $F(x_{A \cup B \cup C}) - F(x_{A \cup B}) = 8 - 7 = 1$.

These contributions are again multiplied with different weights depending on how many features that are in S. In this example $|S| \in \{0, 1, 2\}$, and as a result, the weights are $\{\frac{1}{3}, \frac{1}{6}, \frac{1}{3}\}$. There are now four different ways for assessing feature A, these are $\{0 \leftarrow A, B \leftarrow A, C \leftarrow A, B \cup C \leftarrow A\}$. This results in that the Shapley value for A becomes,

$$\phi_A = \frac{5}{3} + \frac{4}{6} + \frac{4}{6} + \frac{8}{3} = \frac{17}{3}.$$

Notice that there are only four parts in the equation above, while there are six numbers beneath Feature A in Table 3.1. This is because there is a repetition in two of the processes that creates these values. One could just take the sum of every contribution and divide it by 6 to get the same Shapley values as the method described above. However, as the amount of features increases, this will be very computational expensive, and as a result the method above is the preferred one.

Table 3.1: An example of how Shapley values are calculated

	Feature A	Feature B	Feature C
A ← B ← C	5	2	1
A ← C ← B	5	5	-2
B ← A ← C	4	3	1
B ← C ← A	8	3	-3
C ← A ← B	4	5	-1
C ← B ← A	8	1	-1
Sum	34	19	-5
SHAP value (ϕ)	$\frac{17}{3}$	$\frac{19}{6}$	$-\frac{5}{6}$

Shap

"shap" is a popular open source python library used for interpreting and explaining machine learning models. It provides a unified framework for understanding the importance and contribution of each feature in a predictive model's output. The primary goal of "shap" is to help users gain insights into their model's predictions and understand the underlying factors driving those predictions. In [30] one can find comprehensive information on shap's implementation of SHAP values. This includes installation instructions, package details and visualization tools.

SHAP values builds on the same principles as the Shapley values, however, finding exact Shapley values for datasets with many features can become computationally impossible to do. SHAP values acts as an approximation of the Shapley values. SHAP values for a single model can be visualized in shap, this visualization can among other show the rank of the features based on their absolute impact on the model output. Other visualizations can show how high and low feature values can impact the predicted output. This is a valuable method to get an understanding of how the features influences a predicting model. However SHAP values also have their drawbacks, they are often very computational costly to produce for advanced predicting models with many features.

3.2 Logistic Regression

Logistic regression is a statistical technique used to examine the relationship between a binary target variable and one or more predictors, which can be either categorical or continuous. Understanding binary outcomes has been a critical aspect of statistical analysis, and logistic regression has been the preferred model for binary regression since 1970 [1]. The model is not computationally heavy, and also allows for a straightforward interpretation of each predictor variable.

3.2.1 Generalized Linear Models

Generalized linear models serve as the basis for understanding logistic regression. These models extend the concept of ordinary linear regression by linking the mean of the response variable to the predictors using a link function, while also enabling the variance of each observation to be a function of its mean.

The response variable in generalized linear models is typically represented as y_i, where i ranges from 1 to n, with n being the number of predictors. It is assumed that y follows a specific exponential distribution family, which is characterized by a corresponding probability density function,

$$P(y_i;\theta_i,\phi_i) = h(y_i,\phi_i)\text{exp}\left(\frac{y_i\theta_i - \kappa(\theta_i)}{\phi_i}\right). \tag{3.14}$$

The exponential distribution family used in generalized linear models consists of several parameters, including the canonical parameter θ_i, the dispersion parameter ϕ_i, the normalizing function $h(y_i, \phi_i)$, and the known second derivative function $\kappa(\theta_i)$.

The mean of the exponential family can be expressed as

$$E(y_i) = \mu_i = \frac{d(\kappa(\theta_i))}{d\theta_i}, \tag{3.15}$$

and the variance as

$$Var(y_i) = \phi_i \frac{d^2(\kappa(\theta_i))}{(d\theta_i)^2}. \tag{3.16}$$

Generalized linear models also include a systematic component known as η, which is defined as

$$\eta_i = \sum_{j=1}^{p} \beta_j x_{ij}, \tag{3.17}$$

where p denotes the number of features in the dataset.

One can use a link function, $g(\mu_i) = \eta_i$, to connect η with the mean. In generalized linear models, the primary objective is to compute the estimates for β_j. This results in an estimation of $g(\mu)$, which in turn provides an estimate for μ.

Logistic regression involves a binary response, and it assumes that y follows a binomial (or Bernoulli) distribution. One can examine the scenario in which the distribution is binomial,

$$y_i = \text{Bin}(n_i = 1, \pi_i), \tag{3.18}$$

where π_i is the conditional probability, which is defined as

$$\pi_i = E(y_i) = P(y_i = 1 | \boldsymbol{x}_i). \tag{3.19}$$

A generalized linear model for the binomial distribution can be constructed since the response follows a distribution that is part of the exponential family,

$$\boldsymbol{x}_i^T \boldsymbol{\beta} = \eta_i = g(\mu_i) = g(\pi_i). \tag{3.20}$$

The binary logistic model is obtained using a logit link function,

$$g(\pi_i) = \log\left(\frac{\pi_i}{1-\pi_i}\right) = \eta_i = \beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip}. \tag{3.21}$$

The other way around, this can be expressed with the logistic response function

$$\pi_i = \frac{e^{\eta_i}}{1+e^{\eta_i}}. \tag{3.22}$$

3.2.2 Parameter estimation

To achieve desirable outcomes with logistic regression, it is crucial to have good estimates of the regression parameters $\boldsymbol{\beta} = (\beta_0, \beta_1, ..., \beta_k)$ [26]. The maximum likelihood approach is often

employed to accomplish this. The likelihood function is formulated by taking the product of the probability densities of y_i,

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{n} f(y_i|\boldsymbol{\beta}) = \prod_{i=1}^{n} \pi_i^{y_i}(1-\pi_i)^{1-y_i}. \tag{3.23}$$

As $\boldsymbol{\beta}$ has a direct influence on the likelihood function, determining the values of the β_j's can be achieved by maximizing the likelihood function. However, the process of maximizing the likelihood function can be very complex, thus it is often preferred to maximize the logarithm of the likelihood function instead,

$$\log(L(\boldsymbol{\beta})) = l(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(y_i \log\left(\frac{\pi_i}{1-\pi_i}\right) + \log(1-\pi_i) \right). \tag{3.24}$$

Due to the fact that

$$1 - \pi_i = \frac{1}{1+\exp(\boldsymbol{x}_i^T\boldsymbol{\beta})}, \tag{3.25}$$

one get that the logarithm of the likelihood function can be expressed as

$$l(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(y_i \boldsymbol{x}_i^T \boldsymbol{\beta} - \log(1 + exp(\boldsymbol{x}_i^T\boldsymbol{\beta})) \right). \tag{3.26}$$

A common technique when seeking the maximum of a differentiable function is to locate points where its derivative is zero. This approach is also employed in this scenario, where the aim is to determine the values of β_j that lead to a score function of $s(\boldsymbol{\beta}) = 0$. The score function is defined as,

$$s(\boldsymbol{\beta}) = \frac{\partial l(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}}. \tag{3.27}$$

Solving $s(\boldsymbol{\beta}) = 0$ involves a multi-dimensional system of non-linear equations. In order to obtain the solutions to these equations, a numerical method is necessary. To facilitate this process, the observed Fisher information $H(\boldsymbol{\beta})$ is introduced,

$$H(\boldsymbol{\beta}) = -\frac{\partial s(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}^T}. \tag{3.28}$$

By utilizing this information, the numerical solution to the set of equations can be obtained in an efficient and accurate manner. The Fisher scoring algorithm or the Newton-Raphson method are commonly the methods of choice. In this situation, these methods coincide since the observed Fisher information $H(\boldsymbol{\beta})$ is equal to the expected Fisher information $F(\boldsymbol{\beta})$. As a result, the process for determining the regression parameters $\boldsymbol{\beta}$ is as follows: by initializing the value of $\boldsymbol{\beta}$, the score function $s(\boldsymbol{\beta})$ is computed, and then the observed Fisher information matrix $H(\boldsymbol{\beta})$ is obtained. By applying the Fisher scoring algorithm or the Newton-Raphson method, an updated value of $\boldsymbol{\beta}$ is obtained using this equation,

$$\hat{\boldsymbol{\beta}}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} + H^{-1}(\hat{\boldsymbol{\beta}}^{(t)})s(\hat{\boldsymbol{\beta}}^{(t)}). \tag{3.29}$$

This process is repeated until convergence is achieved, and the final value of $\boldsymbol{\beta}$ is deemed to be the optimal solution.

The positivity and invertibility of the expected Fisher information ensure that the observed Fisher information is also positive definite. This property, coupled with the concavity of the log-likelihood function, implies that the maximum-likelihood estimates are always unique.

3.2.3 L1 and L2 Regularization

L1 and L2 regularization are widely used techniques to simplify models that have a large number of features in the dataset. These regularization methods are employed to counteract overfitting and facilitate feature selection. When applied to logistic regression, L1 regularization is referred to as lasso regression, while L2 regularization is known as ridge regression.

Lasso regression and ridge regression operate by introducing a penalty term into the log likelihood function. When there are p features present in the dataset, the log likelihood function for lasso regression becomes

$$l(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(y_i \boldsymbol{x}_i^T \boldsymbol{\beta} - \log(1 + \exp(\boldsymbol{x}_i^T \boldsymbol{\beta})) \right) - \lambda \sum_{j=1}^{p} |\beta_j|, \tag{3.30}$$

and the log likelihood function adjusted for ridge regression becomes

$$l(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(y_i \boldsymbol{x}_i^T \boldsymbol{\beta} - \log(1 + \exp(\boldsymbol{x}_i^T \boldsymbol{\beta})) \right) - \lambda \sum_{j=1}^{p} \beta_j^2. \tag{3.31}$$

In each of these equations, λ represents a hyperparameter that can be adjusted, and as λ increases in value, a greater penalty is imposed on model complexity.

The primary distinction between lasso regression and ridge regression lies in their effects on the coefficients $\boldsymbol{\beta}$. Specifically, lasso regression shrinks the coefficients of less significant features to zero, effectively eliminating them from consideration. As a consequence, lasso regression is well-suited for feature selection. Ridge regression on the other hand usually shrinks the coefficients to a lower value, but never to zero. A combination between ridge and lasso regression is sometimes used.

3.2.4 Scikit-learn

Scikit-learn, also known as sklearn, is a popular machine learning library for Python. It is an open-source software package that is built on top of NumPy, SciPy, and Matplotlib [43]. Scikit-learn was initially released in 2007, and provides a wide range of tools for data pre-processing, feature engineering, model selection, and model evaluation.

Scikit-learn is designed to be easy to use and comes with extensive documentation, tutorials, and examples. It supports a wide range of machine learning algorithms, including linear regression, logistic regression, decision trees, random forests, k-nearest neighbors, support vector machines, and neural networks. These algorithms can be used for classification, regression, clustering, and dimensionality reduction tasks.

Scikit-learn has become the go-to library for many machine learning practitioners and researchers. It is widely used in industry and academia for a wide range of applications, including natural language processing, image classification, and recommendation systems. Its popularity is due to its ease of use, efficiency, and rich functionality, as well as its extensive community support.

Scikit-learn provides a powerful implementation of logistic regression that is both easy to use and highly flexible. In [37] one can find comprehensive information on Scikit-learn implementation of logistic regression. This includes installation instructions, package details and hyperparameter

information. Additionally, it provides some insights into the mathematics behind the different algorithms.

3.3 Gradient Boosted Decision Trees

Gradient boosted decision trees has shown impressive success in various practical applications, including but not limited to regression and classification problems, as it can effectively handle complex non-linear relationships between variables. Its ability to combine multiple simpler decision trees in one prediction, while minimizing a loss function makes it a powerful tool for predictive modeling in the field of machine learning.

3.3.1 Decision Trees

Tree-based learning methods are widely used and considered to be among the best supervised learning models [2]. They are versatile and can be used for both regression and classification problems. Tree-based methods are known for their high accuracy, stability, and ability to handle non-linear data.

At the heart of all tree-based learning methods are decision trees, which can be classified into regression trees and classification trees. A classification tree is used when the response variable is categorical, while a regression tree is used when the response variable is continuous. The focus of this thesis is on binary classification trees.

A decision tree is a data structure that represents a way of traversing a dataset. The tree structure guides the data to an expected outcome based on control statements or values. This is done so that different data points lie on either side of a splitting node, depending on the values of a specific feature. Once the decision tree is constructed, it can be used to make predictions for new data points by traversing the tree and classifying them based on the leaf node they end up in. This procedure is illustrated in Figure 3.4.

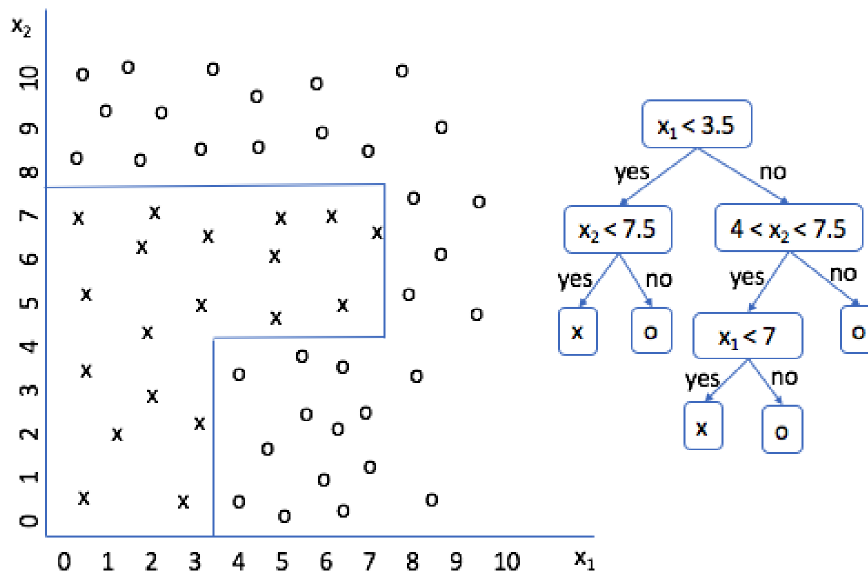

Figure 3.4: A visual representation that demonstrates how a decision tree can perform binary classification. Taken from [18].

Attribute Selection Measure

In a decision tree algorithm, the attribute selection measure employed determines how each split will be made. The splitting rules specify how the data will be divided at each level of the tree. The attribute that yields the best score for a specific measure is chosen as the splitting attribute. The two most commonly used attribute selection measures used for classification are entropy and the Gini Index.

Let p_k represent the fraction of the training data belonging to the kth class. For example, if a node has 3 zeroes and 5 ones, then p_0 would be $\frac{3}{8}$, and p_1 would be $\frac{5}{8}$.

For the entropy selection measure, the entropy computation becomes

$$E = -p_0 \log_2(p_0) - p_1 \log_2(p_1). \tag{3.32}$$

The entropy selection measure is designed such that the resulting value E, falls within the interval $[0, 1]$. A value of $E = 1$ indicates that there is no order present, meaning that $p_0 = p_1 = \frac{1}{2}$. In contrast, $E = 0$ implies that there is maximum order, and either $p_0 = 0$ and $p_1 = 1$, or $p_0 = 1$ and $p_1 = 0$.

For the Gini index selection measure, the definition becomes

$$G = 1 - p_0^2 - p_1^2. \tag{3.33}$$

The Gini index is designed such that the resulting value G, falls within the interval $[0, \frac{1}{2}]$. A value of $G = \frac{1}{2}$ means that $p_0 = p_1 = \frac{1}{2}$. In contrast, $G = 0$ implies that p_0 and p_1 are totally separated, and either $p_0 = 0$ and $p_1 = 1$, or $p_0 = 1$ and $p_1 = 0$. Just like entropy, the objective of using the Gini index is to minimize its value.

While entropy is mainly used for binary classification problems, the Gini index is commonly used for both binary and multi-class classification problems. The Gini index measures the impurity of the classes in a given node by calculating the total variance across all classes.

When a node has $E = 0$ or $G = 0$, it means that there is only one class in that node, and it becomes a leaf node. Because there is no need for further branching, the decision tree terminates at that point.

To avoid considering every possible tree that is possible to build, a greedy approach called recursive binary splitting is commonly used. In this approach, each node is successively split into two new nodes, starting from the top of the tree. At each node, the best split is chosen by considering all predictors and all possible cutpoints for each of the predictors. This greedy approach simplifies the search for the best tree and results in a decision tree that is close to optimal for the given data.

Tree Regularization

The greedy algorithm is a popular method for constructing decision trees. However, this method may result in trees that are large and complex. Unless some datapoints are overlapping on every single predictor, but not on the response, it is possible to construct a tree that perfectly separates each class. However, such a model is prone to overfitting and may not perform well when applied on new data.

To prevent overfitting, it is important to use techniques that reduce the complexity of the tree. One popular technique is known as pruning, which involves removing branches that do not contribute much to the accuracy of the tree. This leads to the creation of a smaller, simpler tree that is less likely to overfit and hence will perform better on new data. In this way one creates a subtree of the original tree $T \subset T_0$. There are various methods to perform pruning, but they all share the same goal; to minimize a combination of a loss function $\mathbb{L}$ of the predicted and estimated responses, and a penalty term that takes into account the complexity of the model,

$$\mathbb{L}(y, \hat{y}) + \gamma|T|. \tag{3.34}$$

γ is here a regularizing hyperparameter that chooses at what degree complexity is penalized. $|T|$ is the number of leaf nodes of the tree T. As a result, if $\gamma = 0$, then the subtree will be equal to the original tree. As the value of γ increases, the penalty for model complexity becomes stronger, leading to more aggressive pruning of the decision tree. This results in a smaller and simpler subtree.

There are other ways to avoid overfitting as well, such as setting a maximum depth of the tree, or requiring a minimum number of samples to split a node. In more advanced tree-based learning models it is common to use a combination of many regularization methods.

There are some potential drawbacks associated with tree-based methods, [42]. One of the primary concerns is the risk of overfitting. Another issue is the instability of the predictions, as small changes in the training data can result in significantly different trees and hence different predictions. Despite these limitations, tree-based methods can become a powerful tool when used appropriately.

3.3.2 Ensemble learning

Historically, the standard approach to data driven modeling was to build a single, complex model that aimed to accurately predict the target variable. However, in recent years, ensemble learning has become a popular alternative. Ensemble learning involves building multiple, simpler models, known as weak learners, and combining their predictions to create one single more accurate model. One common technique in ensemble learning is bootstrap aggregation.

Bootstrap aggregation

Bootstrap aggregation, also referred to as bagging, is a powerful technique that enhances stability and accuracy in machine learning models, particularly in tree-based models. In bagging, multiple models are trained using different samples from the original dataset.

Given a standard training set D of size n, bagging involves the following three steps:

1. Generate m new training sets $[D_1, \dots, D_m]$ by sampling n observations from D uniformly and with replacement. This means that the same observations can occur more than once in each new training set D_i.

2. Each weak learner in the ensemble is trained on a different bootstrap sample, D_i.

3. The predictions of the individual models are combined to obtain a final prediction. How this is done changes based on what ensemble learning method that is being used.

Bagging is a special case of model averaging, which involves training multiple models and combining their predictions. However, in bagging, each model is trained on a bootstrap sample, whereas in model averaging, each model is trained on the entire dataset. By training each model on a different bootstrap sample, each weak learner will be forced to train a little different, bagging can thus reduce variance and prevent overfitting. Additionally, bagging can improve the stability of the model by reducing the impact of outliers and noisy data points [4].

3.3.3 Gradient Boosting

Gradient boosting is a machine learning technique that creates a predictive model by combining multiple weak prediction models. These weak models are usually simple, such as decision trees with limited depth or linear regression models.

At each iteration of the gradient boosting algorithm, a new weak learner is added to the ensemble of models. The weak learner is trained to minimize the error of the whole ensemble generated so far. This process continues until a stopping criteria is met, such as the error of the ensemble cannot be further minimized or a predefined number of iterations is reached.

The new weak learners are designed to be correlated with the negative gradient of the loss function of the entire ensemble. This approach helps in minimizing the loss function and improving the accuracy of the predictive model. The ensemble of weak learners is combined in such a way that each model contributes to the prediction according to its relative performance. The final prediction is obtained by combining the predictions of all the weak learners in the ensemble.

Mathematically, the objective is to determine a function $\hat{F}(\boldsymbol{x})$ that accurately predicts the response variable based on the values of the explanatory variables. This is accomplished by introducing a loss function $\mathbb{L}(y, F(\boldsymbol{x}))$ and minimizing it in expectation,

$$\hat{F} = \arg\min_{F} \mathbb{E}_{\boldsymbol{x},y}[\mathbb{L}(y, F(\boldsymbol{x}))]. \tag{3.35}$$

There are two distinct approaches to minimizing the expected loss function, and those are numerical optimization and optimization in function space. However, this thesis will exclusively be about optimization in the function space.

Optimization in Function Space

The aim of optimization in function space is to search for functions that can be combined using additive measures to form a function estimate, denoted as $\hat{F}$. Figure 3.5 illustrates the process of constructing $\hat{F}$. The initial function $\hat{F}_0$ is first initialized, followed by a series of boosting steps.

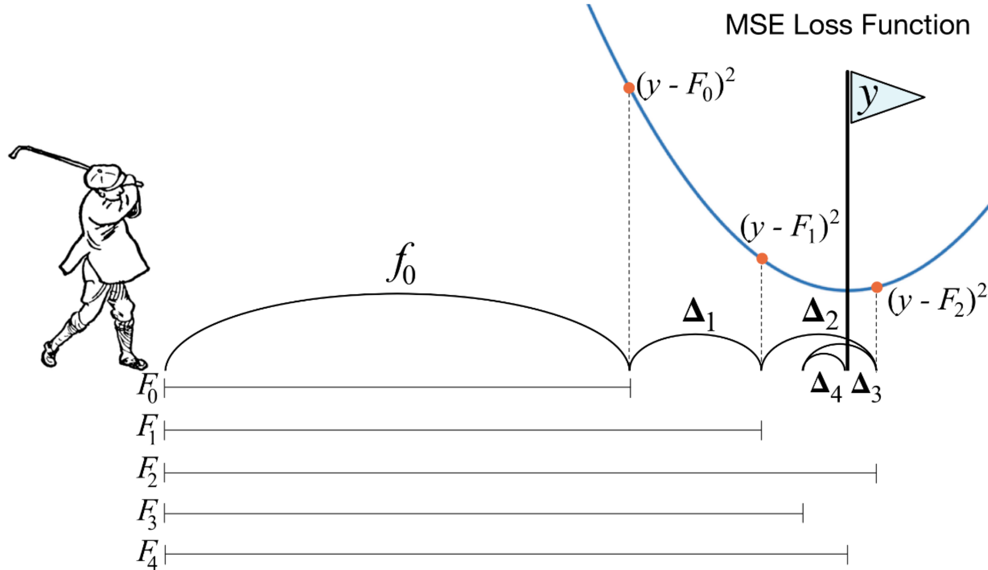

Figure 3.5: A visual representation that shows how every step in boosting can be compared to hitting on a golf ball, gradually moving it closer to the target. Taken from [34].

To understand the basic principles of gradient boosting, let us consider a single observation, where the response variable is denoted y, and the explanatory variables are denoted $\boldsymbol{x}$. Additionally, the weak learners are denoted h_m. Various types of weak learners exist, but common choices include splines or decision trees. An additional parameter, known as the step size or learning rate, denoted as ρ_m, must also be introduced. This parameter must be specified at each iteration. As a result, the function at the m-th iteration can be expressed as

$$\hat{F}_m = \hat{F}_{m-1} + \rho_m h_m, \tag{3.36}$$

where

$$\rho_m h_m = \arg\min_{\rho,h} \left(\mathbb{L}(y, \hat{F}_{m-1}(\boldsymbol{x})) + \rho_m h_m(\boldsymbol{x}) \right). \tag{3.37}$$

The primary goal of gradient boosting is to create a strong learner by iteratively adding weak learners to the model. To achieve this goal, a strategy is employed in which the following principles are central to every gradient boosting algorithm.

The first step is to train a weak learner F_0 to fit the target value y. Remember that these equations only focuses on one single observation.

Subsequently, the following four steps are repeated for M iterations:

1. Compute the gradient g_m using the current ensemble of functions F_{m-1},

$$g_m = E_y \left[\frac{\partial(\mathbb{L}(y, F_{m-1}(\boldsymbol{x})))}{\partial(F_{m-1}(\boldsymbol{x}))} \right]. \tag{3.38}$$

For instance, suppose that the loss function $\mathbb{L}$ is the mean squared error.

$$\mathbb{L}_{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - F(\boldsymbol{x}_i))^2. \tag{3.39}$$

Subsequently, the gradient g_m in our example, can be expressed as

$$g_m = E_y \left[\frac{\partial(\mathbb{L}_{MSE}(y, F_{m-1}(\boldsymbol{x})))}{\partial(F_{m-1}(\boldsymbol{x}))} \right] = 2(y - F_{m-1}(\boldsymbol{x})). \tag{3.40}$$

2. In practice, the negative of the gradient, $-g_m$, can be viewed as a set of pseudo-residuals. These residuals are used to train a weak learner $h_m(\boldsymbol{x})$, such that the weak learner is as equal to the negative of the gradient as possible,

$$h_m(\boldsymbol{x}) = -g_m + \epsilon, \tag{3.41}$$

where ϵ denotes the error between the weak learner and the negative of the gradient.

3. Next, a step size ρ_m is selected such that

$$\rho_m = \arg\min_{\rho} \mathbb{L}(y, F_{m-1}(\boldsymbol{x}) + \rho_m h_m(\boldsymbol{x}))). \tag{3.42}$$

4. Finally, the model $\rho_m h_m(\boldsymbol{x})$ is added to the ensemble.

$$F_m = F_{m-1} + \rho_m h_m. \tag{3.43}$$

3.3.4 Gradient Boosted Decision Trees

Gradient boosted decision trees are a popular form of gradient boosting that has demonstrated state-of-the-art results in various real-world problems. This approach is an excellent way of capturing interactions between explanatory variables. The objective of gradient boosted decision trees is to minimize a loss function by adding new decision trees iterative as shown in figure 3.6.

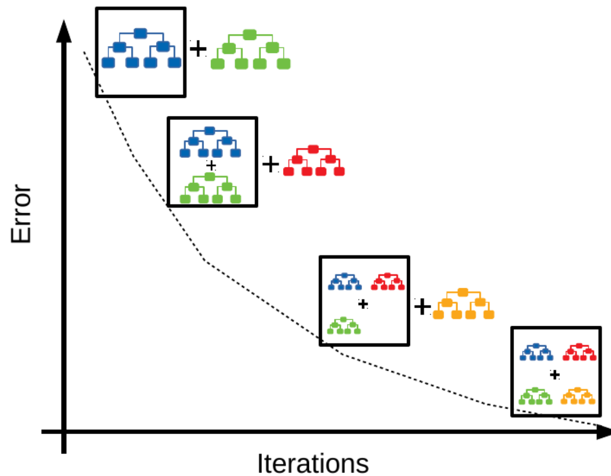

Figure 3.6: A visual representation that shows the main idea behind gradient boosted decision trees. As each tree is incorporated into the overall function in gradient boosted decision trees, the error decreases. Taken from [33].

In each iteration, the new decision tree is built to fit the negative gradient of the loss function. The algorithm uses a greedy approach to find the best split at each node, with the aim of minimizing the loss function.

Gradient boosted decision trees can serve for both classification and regression tasks [28]. For classification, the target variable is categorical, and the algorithm aims to minimize the log loss or another suitable classification loss function. As with other gradient boosting algorithms, gradient boosted decision trees can be prone to overfitting if the model is too complex. Therefore, regularization techniques such as shrinkage or early stopping are often used to prevent overfitting.

The algorithm uses an ensemble of tree models that can predict an output based on M additive functions,

$$\hat{y}_i = \sum_{m=1}^{M} F_m(\boldsymbol{x}_i). \tag{3.44}$$

Here y_i and $\boldsymbol{x}_i$ denotes one observation, where $\boldsymbol{x}_i$ is a vector. Each tree F_m follows the same structure as the decision trees explained earlier in this section. However, in gradient boosted decision trees, each leaf node is assigned a weight w. This weight is used to calculate the final prediction by summing the weights in the corresponding leaves.

Just like in the subsection before, the primary goal is to minimize a regularized objective [15]. Here, the regularized objective is defined as

$$\text{obj} = \sum_{i=1}^{n} \mathbb{L}(y_i, \hat{y}_i^{(t)}) + \sum_{i=t}^{t} \Omega(F_i). \tag{3.45}$$

The i-th observation's prediction in the t-th iteration is denoted by $\hat{y}_i^{(t)}$. The loss function $\mathbb{L}$ measures the deviation of predicted values from actual values, using a chosen metric. The second term, Ω, penalizes model complexity and can be defined as follows

$$\Omega(F) = \gamma|T| + \frac{1}{2}\lambda||\boldsymbol{w}||^2. \tag{3.46}$$

γ is here a regularizing hyperparameter that chooses how much complexity is penalized. $|T|$ is the

number of leaf nodes of the tree T. In the second part, λ is a shrinkage hyperparameter. The larger the value of λ, the more the model's sensitivity to individual observations is reduced.

Since $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + F_t(\boldsymbol{x}_i)$, the regularized objective in the t-th iteration can be expressed as

$$\text{obj}^{(t)} = \sum_{i=1}^{n} \Big(\mathbb{L}(y_i, \hat{y}_i^{(t-1)} + F_t(\boldsymbol{x}_i)) \Big) + \Omega(F_t). \tag{3.47}$$

Approximating the loss function $\mathbb{L}$ up to the second order using Taylor expansion around $\hat{y}_i^{(t-1)}$ yields the following objective

$$\text{obj}^{(t)} \approx \sum_{i=1}^{n} \left(\mathbb{L}(y_i, \hat{y}_i^{(t-1)}) + g_i F_t(\boldsymbol{x}_i) + \frac{1}{2} h_i F_t^2(\boldsymbol{x}_i) \right) + \Omega(F_t). \tag{3.48}$$

Here g_i and h_i are defined as

$$g_i = \frac{\partial \mathbb{L}(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}$$

$$h_i = \frac{\partial^2 \mathbb{L}(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)} \partial \hat{y}_i^{(t-1)}}.$$

Observe that the term $\mathbb{L}(y_i, \hat{y}_i^{(t-1)})$ is independent of $F_t(\boldsymbol{x}_i)$, and since the objective is to optimize $F_t(\boldsymbol{x}_i)$, any terms that do not depend on $F_t(\boldsymbol{x}_i)$ can be eliminated, resulting in a new objective,

$$\text{obj}^{(t)} = \sum_{i=1}^{n} \left(g_i F_t(\boldsymbol{x}_i) + \frac{1}{2} h_i F_t^2(\boldsymbol{x}_i) \right) + \Omega(F_t). \tag{3.49}$$

Before proceeding, it will be helpful to refine the definition of a tree $F_t(\boldsymbol{x})$ as the following:

$$F_t(\boldsymbol{x}) = \boldsymbol{w}_{q(\boldsymbol{x})} \qquad \boldsymbol{w} \in R^{|T|} \qquad q : R^d \to \{1, 2, ..., |T|\} \tag{3.50}$$

In this context, the vector $\boldsymbol{w}$ represents the weights assigned to the leaves of the tree. The function q maps each data point to the leaf node it corresponds to, and d is the number of explanatory variables in the dataset. Additionally, $|T|$ still refers to the number of leaves present in the tree.

The reformulated tree model allows one to define the set of observations that belong to node j as $I_j = \{i | q(\boldsymbol{x}_i) = j\}$. The predicted value for all observations within I_j is then given by $F_t(\boldsymbol{x}_i) = w_j$, because w_j represents the score assigned to the corresponding leaf node. This means that the objective value for the t'th tree can be expressed as follows

$$\begin{aligned} \text{obj}^{(t)} &= \sum_{i=1}^{n} \left(g_i w_{q(\boldsymbol{x}_i)} + \frac{1}{2} h_i w_{q(\boldsymbol{x}_i)}^2 \right) + \gamma |T| + \frac{1}{2} \lambda \sum_{j=1}^{|T|} w_j^2 \\ &= \sum_{j=1}^{|T|} \left[(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2 \right] + \gamma |T| \end{aligned} \tag{3.51}$$

Observe that in the second line of the expression, the summation index has been changed because all data points belonging to the same leaf are assigned the same score. For the sake of simplicity, the expression can be further compressed,

$$G_j = \sum_{i \in I_j} g_i \quad \text{and} \quad H_j = \sum_{i \in I_j} h_i,$$

resulting in

$$\text{obj}^{(t)} = \sum_{j=1}^{|T|} \left[G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2 \right] + \gamma |T| \tag{3.52}$$

The equation shows that the w_j values are independent, allowing for the formation of a quadratic equation to determine the optimal w_j values for a given structure q that yields the best objective,

$$w_j^* = -\frac{G_J}{H_j + \lambda} \tag{3.53}$$

$$\text{obj}^* = -\frac{1}{2} \sum_{j=1}^{|T|} \frac{G_j^2}{H_j + \lambda} + \gamma |T|. \tag{3.54}$$

To gain a better understanding of this concept, Figure 3.7 is provided. Here, it is important to recall that g_i and h_i represent the gradients and Hessians of the loss function. The fundamental idea involves adding these gradients and Hessians together depending on how the tree is structured. By utilizing the objective in equation 3.54, one can now evaluate how good the tree is.

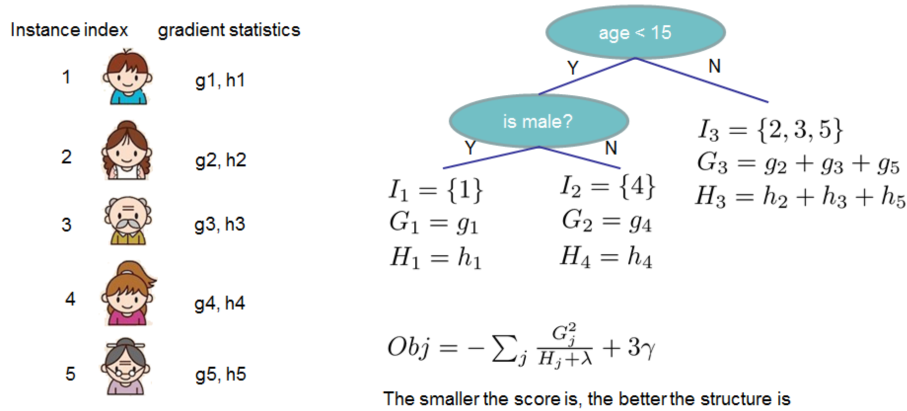

Figure 3.7: An illustration clarifying how the objective score is calculated. For a particular tree structure, the statistics g_i and h_i are assigned to their respective leaves. Then, the formula is employed to compute the tree's quality. Taken from [15].

In an ideal scenario, one would evaluate every conceivable tree and select the best one when constructing a tree. However, in practice, this is a computationally expensive task. Therefore, similar to decision trees, it is common to optimize one level of the tree at a time. This involves splitting a leaf into two leaves, and the gain from such a split will then be equal to

$$\text{Gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_R + G_L)^2}{H_R + H_L + \lambda} \right] - \gamma. \tag{3.55}$$

This equation can be obtained by using equation 3.54, and then multiply with -1. The result in a function where maximizing is the goal. In this equation, L and R denotes the collection of instances in the left and right nodes, after the split. This measure can be leveraged to determine the optimal split by identifying the maximum gain, rather than leaving the leaf unchanged.

3.3.5 L1 and L2 Regularization

L1 and L2 regularization are popular tools for preventing over-fitting and reducing complexity in gradient boosted decision trees. These methods accomplish this by adding a penalty term to the objective that is being minimized. This penalty term penalizes the size of the weights. This results in a new objective, as in equations 3.45 and 3.46, where L2 regularization is done. The reason L2 regularization is included in the introduction to gradient boosted decision trees is because it is a widely used method to tackle the issue of overfitting, which is an often observed problem in this type of predicting models.

However, it is possible to use L1 regularization instead of L2 regularization. For L1, the new penalty term becomes

$$\Omega(F) = \gamma|T| + \lambda||\boldsymbol{w}||. \tag{3.56}$$

For more advanced algorithms, it is common to use a combination of both L1 and L2 regularization, then the new penalty term becomes

$$\Omega(F) = \gamma|T| + \lambda_1||\boldsymbol{w}|| + \frac{1}{2}\lambda_2||\boldsymbol{w}||^2. \tag{3.57}$$

In both these equations λ is a tuneable hyperparameter, and the larger λ is, the more complexity is penalized.

The key difference between L1 and L2 regularization is that L1 regularization tends to shrink many of the weights to zero, hence creating sparse models. L2 regularization on the other hand tends to create models with lower values, but not zero, for the weights.

It is important to know that changing the regularization changes the objective, and as a result, the end objective will not be the same as derived in the subsection above.

3.3.6 LightGBM

LightGBM, short for Light Gradient-Boosting Machine, is a distributed gradient-boosting framework for machine learning that is available for free and is open-source. Microsoft created it and released it in 2016 [23]. LightGBM is programmed in C++, Python, R, and C and it supports C++, Python, R, and C#. LightGBM is compatible with Windows, macOS, and Linux.

LightGBM is recognized for its ability to perform parallel tree boosting. LightGBM was created after another popular tree boosting algorithm, XGBoost. XGBoost quickly became popular after launch, it was designed to be highly efficient, flexible and portable. XGBoost was made for real-world scale problems, and could solve problems beyond billions of examples with a minimal amount of resources and produce state of the art results [16]. However, XGBoost could be very computational costly. LightGBM was made to produce just as good results as XGBoost, but to handle the training in a much faster way.

LightGBM is recognized for its ability to perform parallel tree boosting. LightGBM shares many of the advantages of XGBoost, such as supporting sparse optimization, multiple loss functions and regularizations, bagging, and early stopping. However, there is a difference between the two in terms of tree construction. Unlike XGBoost, LightGBM does not adopt a level-wise strategy. Instead, it grows trees in a leaf-wise manner and selects the leaf that yields the greatest gain. Additionally, LightGBM does not employ the sorted-based decision tree learning algorithm used by XGBoost, which looks for the best split point among sorted feature values. Instead, LightGBM uses a highly optimized histogram-based decision tree learning algorithm, resulting in significant improvements in efficiency and memory consumption. As a result, LightGBM is considered to be faster than XGBoost while also producing just as good results [7].

In [8], one can find comprehensive information on LightGBM, including installation instructions, package details, hyperparameter introductions, and tutorials. Additionally, [44] provides insights into the mathematics and algorithms used in the histogram-based decision tree learning approach applied in LightGBM.

3.4 Deep Learning

Deep learning is a branch in the field of machine learning that utilizes artificial neural networks, a technique that is inspired by the workings of the human brain. Deep learning has proven to be a highly effective machine learning technique in recent years, with numerous successful applications across a diverse range of fields. Deep learning is thus seeing increased popularity and can be applied to a wide range of areas [31]. Deep learning's ability to handle large and complex data sets, combined with its ability to learn patterns and relationships in data, has made it the preferred method for many machine learning tasks. In this thesis, deep learning will be used for binary classification.

3.4.1 The Artificial Neuron

The artificial neuron is the core building block of a neural network, and is inspired by biological neurons.

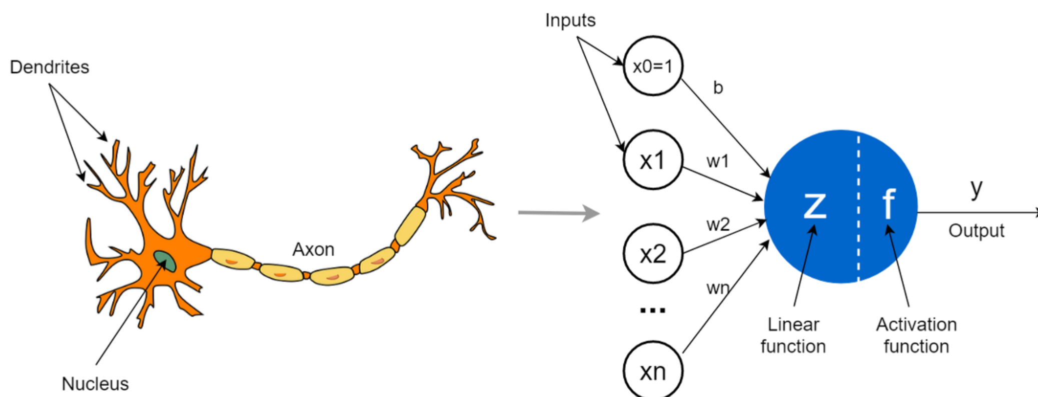

Figure 3.8: A comparison between a biological neuron(left) and an artificial neuron(right). Taken from [36]

.

The artificial neuron has a lot in common with the biological neuron. A biological neuron takes in signals from its neighboring neurons through dendrites, as seen in Figure 3.8. The same goes for an artificial neuron that takes input from other neurons. Biological neurons have synapses connecting to the dendrites, similarly artificial neurons have weights that measure the importance level of each input. In biological neurons, the nucleus processes input from dendrites to create an output, the same is done in the artificial neuron. Lastly both passes this output to a neighboring neuron.

The artificial neuron usually receives information from multiple inputs as showed in Figure 3.8. The inputs x_i are multiplied with each corresponding weight w_i and then added together with a bias term b to form z,

$$z = \sum_{i=1}^{n} (w_i x_i) + b. \tag{3.58}$$

The bias term is individual to each neuron and the purpose of including a bias term is to shift the value of z, this will decrease or increase the output of each individual neuron. The output h that each neuron sends out is a function of z. This function is called an activation function, and is defined

$$\sigma(z) = h. \tag{3.59}$$

Activation functions are a type of transfer functions that determine which information is passed through a neural network. They are used to determine how a neuron should propagate learning data during the learning phase. The activation function decides how "excited" the neuron is, and then passes along the information. There are many choices for what activation function to use, but the most common ones are ReLU, sigmoid, tanh and leakyReLU, these will be discussed later.

3.4.2 The Neural Network

An artificial neural network is comprised of multiple neurons organized into layers. This configuration includes an input layer for data input, one or more hidden layers for processing, and an output layer for producing the final results. This is illustrated in Figure 3.9.

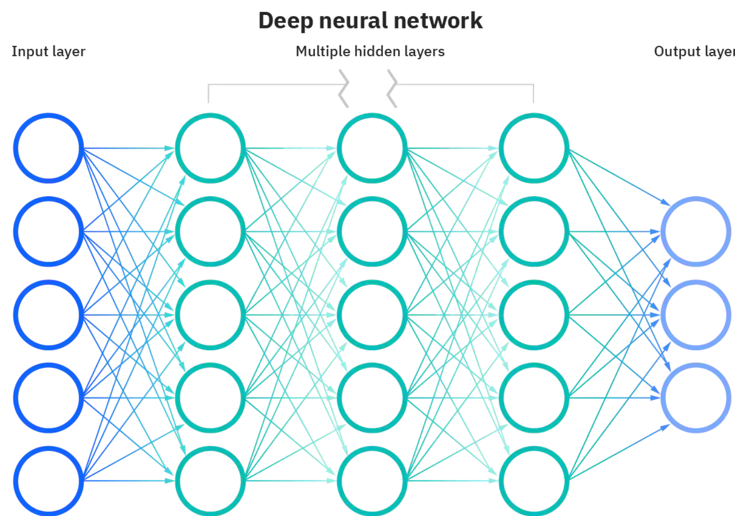

Figure 3.9: An illustration of a neural network. Taken from [17]

.

Having introduced the architecture of the neural network, it's time to take a closer look at the individual layers workings.

The input layer consists of the data $\boldsymbol{x}$, and the number of neurons in this layer will be exactly the same as the number of features in the dataset.

The configuration of the hidden layers, including the number of layers and the number of neurons per layer is a choice where L denotes the number of layers and l denotes the number of neurons per layer. Let $h_l^{(L)}$ denote the value that each neuron sends out. For the first hidden layer (Layer 0), one generates $l_0 + 1$ hidden outputs from $n + 1$ inputs

$$h_0^{(0)} = \sigma\Big(b_0^{(0)} + w_{00}^{(0)} x_0 + w_{01}^{(0)} x_1 + \ldots + w_{0n}^{(0)} x_n\Big)$$

$$h_1^{(0)} = \sigma\Big(b_1^{(0)} + w_{10}^{(0)} x_0 + w_{11}^{(0)} x_1 + \ldots + w_{1n}^{(0)} x_n\Big)$$

$$\vdots$$

$$h_{l_0}^{(0)} = \sigma\Big(b_{l_0}^{(0)} + w_{l_0 0}^{(0)} x_0 + w_{l_0 1}^{(0)} x_1 + \ldots + w_{l_0 n}^{(0)} x_n\Big) \tag{3.60}$$

Remember that b_i is the bias in each neuron.

For the second hidden layer (Layer 1), one generates $l_1 + 1$ hidden outputs from $l_0 + 1$ hidden outputs from Layer 0

$$h_0^{(1)} = \sigma\Big(b_0^{(1)} + w_{00}^{(1)} h_0^{(0)} + w_{01}^{(1)} h_1^{(0)} + \ldots + w_{0l_0}^{(1)} h_{l_0}^{(0)}\Big)$$

$$h_1^{(1)} = \sigma\Big(b_1^{(1)} + w_{10}^{(1)} h_0^{(0)} + w_{11}^{(1)} h_1^{(0)} + \ldots + w_{1l_0}^{(1)} h_{l_0}^{(0)}\Big)$$

$$\vdots$$

$$h_{l_1}^{(1)} = \sigma\Big(b_{l_1}^{(1)} + w_{l_1 0}^{(1)} h_0^{(0)} + w_{l_1 1}^{(1)} h_1^{(0)} + \ldots + w_{l_1 l_0}^{(1)} h_{l_0}^{(0)}\Big) \tag{3.61}$$

Every hidden layer after Layer 1 follows the same principles except the last hidden layer (Layer L). It generates $m + 1$ visible outputs from the information from Layer $L - 1$

$$h_0^{(L)} = \sigma\Big(b_0^{(L)} + w_{00}^{(L)} h_0^{(L-1)} + w_{01}^{(L)} h_1^{(L-1)} + \ldots + w_{0l_{L-1}}^{(L)} h_{l_{L-1}}^{(L-1)}\Big)$$

$$h_1^{(L)} = \sigma\Big(b_1^{(L)} + w_{10}^{(L)} h_0^{(L-1)} + w_{11}^{(L)} h_1^{(L-1)} + \ldots + w_{1l_{L-1}}^{(L)} h_{l_{L-1}}^{(L-1)}\Big)$$

$$\vdots$$

$$h_m^{(L)} = \sigma\Big(b_m^{(L)} + w_{m0}^{(L)} h_0^{(L-1)} + w_{m1}^{(L)} h_1^{(L-1)} + \ldots + w_{ml_{L-1}}^{(L)} h_{l_{L-1}}^{(L-1)}\Big) \tag{3.62}$$

These equations combined with the neural network that they are assign to are illustrated in Figure 3.10.

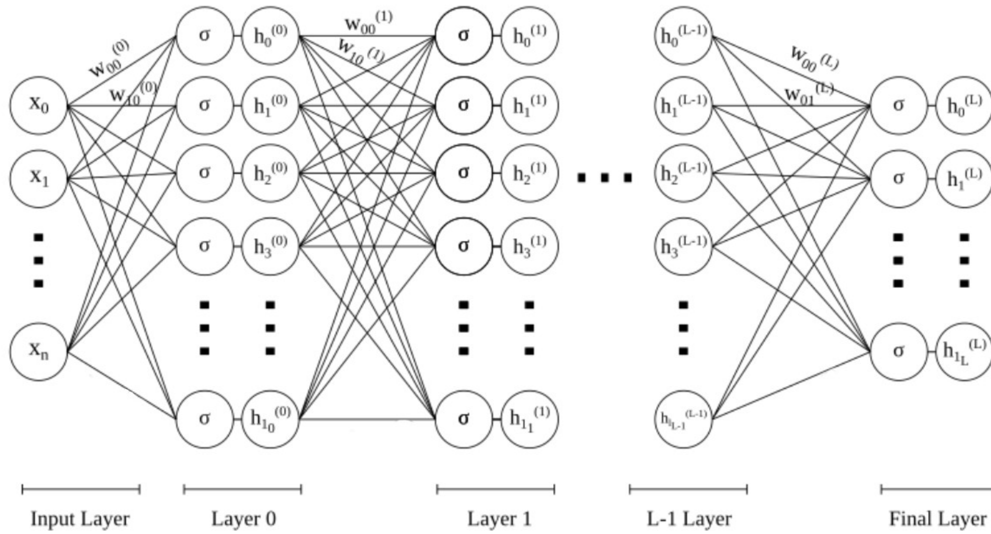

Figure 3.10: An illustration of a multi layered neural network based of the equations above. Notice that the bias term is not included for the sake of simplicity. The illustration is inspired by [6].

3.4.3 Activation functions

Activation functions are an essential component of neural networks. These functions determine the output of a neuron, which is then passed on to the next layer of the network. In this way, activation functions are responsible for transforming the input signal into an output signal that can be used for further processing. In this subsection, some of the most commonly used activation functions in neural networks will be discussed.

The sigmoid function is a special form of the logistic function, and is defined

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{3.63}$$

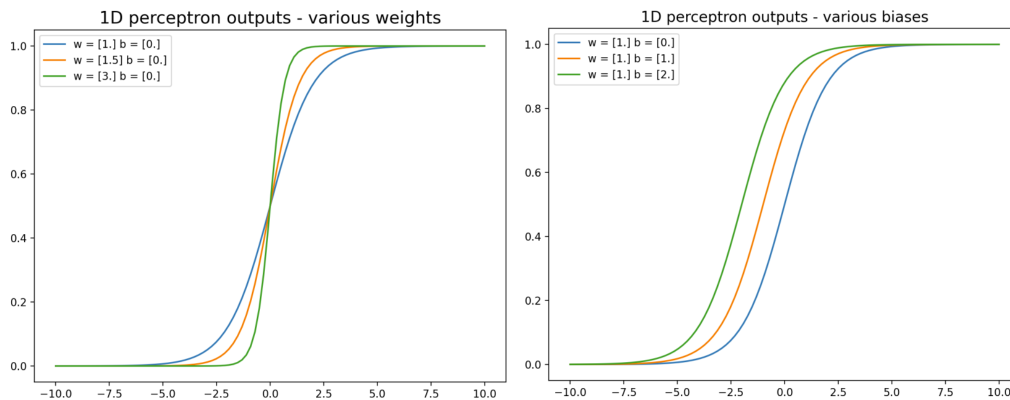

Figure 3.11: An illustration of sigmoid curves corresponding to various parameter values where $z = wx + b$. On the left one can see how the values of the weights influence the steepness of sigmoid function. On the right one can see how the bias shifts the function left or right. The figures are inspired by [6].

The sigmoid function has the property that it will map the entire number line into between 0 and 1. Using this function for deep learning was inspired by the activation potential in biological neural networks. Sigmoid functions play a crucial role in various machine learning applications where

they are often used to convert real-valued outputs into probabilities. Additionally, the sigmoid function is easily differentiable, which makes it suitable for use in gradient descent algorithms for training neural networks. Figure 3.11 shows how changing the weights and the bias will influence the sigmoid function in the case that $z = wx + b$.

The sigmoid function also has some drawbacks. Firstly, the sigmoid function becomes saturated for large positive or negative numbers. When training each model, this will result in that the gradient in these regions becomes nearly insignificant, hindering the network's ability to learn. Secondly, the outputs of the sigmoid function are not centered at zero, leading to potential issues with the gradient updates for the weights. This can cause undesirable fluctuations in the gradient updates, affecting the stability of the learning process.

An activation function similar to the the sigmoid function is the tanh function. The tanh function is defined

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{3.64}$$

The tanh function is as well as the sigmoid function also easily differentiable, which is desirable. There are mainly two ways in which the tanh function is preferred over the sigmoid function. Firstly, the tanh function is zero-centered, and it has a range of output values ranging from -1 to 1. This can make it useful for applications where negative output values are meaningful, such as in image processing or audio analysis. Secondly as showed in Figure 3.12, the derivative of the tanh function is much steeper than the derivative of the sigmoid function. This characteristic implies that the gradients in the tanh function are much stronger, facilitating faster convergence during training.

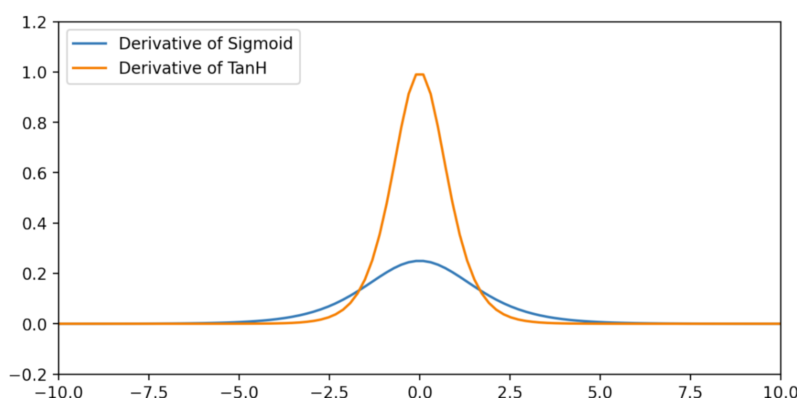

Figure 3.12: An illustration showing the derivative of the sigmoid function and the derivative of the tanh function. The figure is inspired by [6].

A problem that is also present when working with the tanh function, is that this function also becomes saturated for large positive and negative numbers.

Another activation function is Rectified Linear Unit or ReLU for short. It is a simple yet powerful function that has shown enhancements in the performance of neural networks. ReLU operates by outputting the input value if it's positive, and zero if it's negative. This function has proven to be effective in many deep learning applications, as it promotes sparsity in the network and reduces the likelihood of overfitting. ReLU also has the advantage that for positive values, the function never gets saturated.

A variant of the ReLU activation function that is commonly used in deep learning is the leakyReLU. In contrast to the standard ReLU, the leakyReLU has a small slope for negative input values, which allows for a non-zero gradient even for negative inputs. This can help with the "dying ReLU" problem, where ReLU units become inactive and produce zero gradients. The slope coefficient of the leakyReLU is set before training. Both the ReLU and the leakyReLU can be defined as.

$$f(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha z & \text{if } z < 0 \end{cases} \tag{3.65}$$

For ReLU α is set to 0. The ReLU and leakyReLU functions are not smooth unless $\alpha = 1$, meaning they have a sharp corner at zero.

Choosing the right activation function can significantly affect the performance of the network. Each activation function has its advantages and disadvantages, and the choice of function depends on the problem being solved and the architecture of the network. It is common that large neural networks contain different activation functions for different layers.

3.4.4 Training the Neural Network

Training a neural network is a complex process that involves adjusting the weights and biases of the network's nodes to minimize the difference between the network's predicted output and the known correct output for a given set of input data. This process is known as learning, and it involves repeatedly feeding the network with training data and adjusting the network's parameters based on the errors it makes.

For simplicity, one can consider the objective to be an arbitrary loss function $\mathbb{L}$ that is differentiable. This objective is now a result of the weights $\boldsymbol{w}$ and the biases $\boldsymbol{b}$, and can be written $\mathbb{L}(\boldsymbol{w}, \boldsymbol{b})$, where

$$\boldsymbol{w} = \left[w_{00}^{(0)}, w_{01}^{(0)}, \ldots, w_{00}^{(1)}, w_{01}^{(1)}, \ldots, w_{00}^{(L)}, w_{01}^{(L)}, \ldots\right],$$

and

$$\boldsymbol{b} = \left[b_{0}^{(0)}, b_{1}^{(0)}, \ldots, b_{0}^{(1)}, b_{1}^{(1)}, \ldots, b_{0}^{(L)}, b_{1}^{(L)}, \ldots\right].$$

As a result, the task will be to find the $\boldsymbol{w}$ and the $\boldsymbol{b}$ that gives the lowest objective. Using the gradients of the objective to continuously update the weights and biases gives these equations

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \rho \nabla_{w_t} \mathbb{L},$$

$$\boldsymbol{b}_{t+1} = \boldsymbol{b}_t - \rho \nabla_{b_t} \mathbb{L}. \tag{3.66}$$

Here ρ is a stepsize parameter, and t denotes the stepwise update numerator. The equations can be rewritten in terms of the scalar components

$$w_{jk\ (t+1)}^{(l)} = w_{jk\ (t)}^{(l)} - \rho \frac{\partial \mathbb{L}}{\partial w_{jk\ (t)}^{(l)}},$$

$$b_{j\ (t+1)}^{(l)} = b_{j\ (t)}^{(l)} - \rho \frac{\partial \mathbb{L}}{\partial b_{j\ (t)}^{(l)}}. \tag{3.67}$$

Backpropagation

It turns out that finding a closed form for the equations in 3.67 is very difficult. As a result it is normal to use a technique called backpropagation. This technique works by propagating errors backwards through the network. In this way, backpropagation provides a way to compute

the derivatives of the loss function with respect to each weight and bias parameter, allowing for updates that improves the network.

To grasp the concept of backpropagation, one can first start by looking at a neural network with only a single neuron per layer. This is a big simplification, but it allows for individual weights and biases per layer, such that there is no need for any subscripts other than in what layer they are in. This kind of network is showed in 3.68.

$$x_i \xrightarrow[b^{(0)}]{w^{(0)}} z^{(0)} \xrightarrow{\sigma()} h^{(0)} \xrightarrow[b^{(1)}]{w^{(1)}} z^{(1)} \xrightarrow{\sigma()} h^{(1)} \dots h^{(l-1)} \xrightarrow[b^{(l)}]{w^{(l)}} z^{(l)} \xrightarrow{\sigma()} h^{(l)} \dots h^{(L-1)} \xrightarrow[b^{(L)}]{w^{(L)}} z^{(L)} \xrightarrow{\sigma()} h^{(L)} \tag{3.68}$$

Where

$$z^{(0)} = w^{(0)} x_i + b^{(0)},$$

$$z^{(l)} = w^{(l)} h^{(l-1)} + b^{(l)} \qquad \text{for} \quad l = 1, 2, \dots, L$$

and

$$h^{(l)} = \sigma\big(z^{(l)}\big) \qquad \text{for} \quad l = 0, 1, \dots, L.$$

Let us now consider a single input output pair, x_i and y_i, and a loss function given as

$$\mathbb{L} = \frac{1}{2}\big(h^{(L)} - y_i\big)^2.$$

To reduce complexity, one can define an auxiliary variable $\delta^{(l)}$,

$$\delta^{(l)} = \frac{\partial \mathbb{L}}{\partial z^{(l)}} \qquad \text{for } l \in \{0, 1, \dots, L\}. \tag{3.69}$$

The partial derivatives of the loss function with respect to the weights and the biases can be written as

$$\frac{\partial \mathbb{L}}{\partial w^{(l)}} = \frac{\partial \mathbb{L}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial w^{(l)}} = \delta^{(l)} \frac{\partial (w^{(l)} h^{(l-1)} + b^{(l)})}{\partial w^{(l)}} = \delta^{(l)} h^{(l-1)}, \tag{3.70}$$

$$\frac{\partial \mathbb{L}}{\partial b^{(l)}} = \frac{\partial \mathbb{L}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b^{(l)}} = \delta^{(l)} \frac{\partial (w^{(l)} h^{(l-1)} + b^{(l)})}{\partial b^{(l)}} = \delta^{(l)}. \tag{3.71}$$

For the auxiliary variable $\delta^{(l)}$, one can again use the chain rule

$$\delta^{(l)} = \frac{\partial \mathbb{L}}{\partial z^{(l)}} = \frac{\partial \mathbb{L}}{\partial z^{(l+1)}} \frac{\partial z^{(l+1)}}{\partial h^{(l)}} \frac{\partial h^{(l)}}{\partial z^{(l)}} = \delta^{(l+1)} \frac{\partial (w^{(l+1)} h^{(l)} + b^{(l+1)})}{\partial h^{(l)}} \frac{d\sigma\big(z^{(l)}\big)}{dz^{(l)}}. \tag{3.72}$$

Let us assume that the activation function is the sigmoid function, where it is known that $\frac{d\sigma(z)}{dz} = \sigma\big(z\big)(1 - \sigma\big(z\big))$. Then $\delta^{(l)}$ can be written as

$$\delta^{(l)} = \delta^{(l+1)} w^{(l+1)} \sigma\big(z^{(l)}\big)(1 - \sigma\big(z^{(l)}\big)) = \delta^{(l+1)} w^{(l+1)} h^{(l)} (1 - h^{(l)}). \tag{3.73}$$

Note that this deriving of $\delta^{(l)}$ works with all the activation functions that are analytically differentiable.

Before one can begin updating all the gradients one first needs to derive the auxiliary variable for the L-layer,

$$\delta^{(L)} = \frac{\partial \mathbb{L}}{\partial z^{(L)}} = \frac{\partial \mathbb{L}}{\partial h^{(L)}} \frac{\partial h^{(L)}}{\partial z^{(L)}} = \frac{1}{2} \frac{\partial\left(\left(h^{(L)} - y_i\right)^2\right)}{\partial h^{(L)}} \frac{d\sigma\left(z^{(L)}\right)}{dz^{(L)}}. \tag{3.74}$$

Again one can use the derivative of the sigmoid function, and get that

$$\delta^{(L)} = \left(h^{(L)} - y_i\right)\sigma\left(z^{(L)}\right)(1 - \sigma\left(z^{(L)}\right)) = \left(h^{(L)} - y_i\right)h^{(L)}(1 - h^{(L)}). \tag{3.75}$$

For the training, one will start with initializing the weights $\boldsymbol{w}$ and the biases $\boldsymbol{b}$. Thereafter the network will figure out every value for $\boldsymbol{h}$ as shown in 3.68. This is called forward propagation. One can now evaluate $\delta^{(L)}$ and through iterative measures evaluate every $\delta^{(l)}$ starting with $\delta^{(L-1)}$ using equation 3.73. In addition to calculating $\delta^{(l)}$, one can simultaneously update the weights and biases using the equations 3.70, and 3.71. The weights $\boldsymbol{w}$ and the biases $\boldsymbol{b}$ are now updated and one can start with forward propagation again, this updating can be done repeatedly until a stopping criteria is met.

Multiple neurons per layer

Backpropagation also works for a neural network with an arbitrary number of neurons per layer, however the mathematics becomes more complicated, and the whole idea is not as intuitive as for only one neuron per layer.

Forward propagation works as described earlier, and for simplicity one can rewrite these equations to a vector format.

$$z_j^{(l)} = \sum_{k=0}^{m}\left(w_{jk}^{(l)} h_k^{(l-1)}\right) + b_j^{(l)} \qquad \Longrightarrow \qquad \boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}$$

$$h_j^{(l)} = \sigma\left(z_j^{(l)}\right) \qquad \Longrightarrow \qquad \boldsymbol{h}^{(l)} = \sigma\left(\boldsymbol{z}^{(l)}\right) \tag{3.76}$$

Where

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{00}^{(l)} & w_{01}^{(l)} & \cdots & w_{0m}^{(l)} \\ w_{10}^{(l)} & w_{11}^{(l)} & \cdots & w_{n1}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n0}^{(l)} & w_{n1}^{(l)} & \dots & w_{nm}^{(l)} \end{bmatrix} \qquad \boldsymbol{b}^{(l)} = \begin{bmatrix} b_0^{(l)} & b_1^{(l)} & \dots & b_n^{(l)} \end{bmatrix}^T$$

and

$$\boldsymbol{h}^{(l)} = \begin{bmatrix} h_0^{(l)} & h_1^{(l)} & \dots & h_n^{(l)} \end{bmatrix}^T \qquad \boldsymbol{h}^{(l-1)} = \begin{bmatrix} h_0^{(l-1)} & h_1^{(l-1)} & \dots & h_m^{(l-1)} \end{bmatrix}^T.$$

For binary classification in neural networks it is normal that the output is one number between 0 and 1, this can be contained in one output neuron. Because the output layer is just one neuron, the loss function $\mathbb{L}$ can be the same as for the network with only one neuron per layer. The auxiliary variable $\delta_j^{(l)}$ is now defined almost the same way, but there are now multiple values per layer,

$$\delta_j^{(l)} = \frac{\partial \mathbb{L}}{\partial z_j^{(l)}} \qquad \text{for } l \in \{0, 1, \ldots, L\} \quad \text{and} \quad j \in \{0, 1, \ldots, J\}. \tag{3.77}$$

Where J is the number of neurons in layer l.

Compared to the simple case of one neuron per layer, understanding the evaluation of $\delta_j^{(l)} = \frac{\partial \mathbb{L}}{\partial z_j^{(l)}}$ in the general case for an arbitrary layer l is much more complex. In the general case, the loss function $\mathbb{L}$ does not have a direct dependence on the inner layer variable $z_j^{(l)}$. Instead, the loss depends on the activations of the last layer, which in turn depend on the previous layer, and so on. The z values in any given layer form a complete dependency set for the loss $\mathbb{L}$, which means that the loss can be expressed solely in terms of these variables, with no other variables needed. This means that the loss can be expressed as $\mathbb{L}\big(z_0^{(l+1)}, z_1^{(l+1)}, z_2^{(l+1)}, \ldots\big)$. Then, by using partial differentiation

$$\delta_j^{(l)} = \frac{\partial \mathbb{L}\big(z_0^{(l+1)}, z_1^{(l+1)}, z_2^{(l+1)}, \ldots\big)}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathbb{L}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathbb{L}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial h_j^{(l)}} \frac{\partial h_j^{(l)}}{\partial z_j^{(l)}}. \tag{3.78}$$

By definition,

$$\frac{\partial \mathbb{L}}{\partial z_k^{(l+1)}} = \delta_k^{(l+1)}. \tag{3.79}$$

While

$$\frac{\partial z_k^{(l+1)}}{\partial h_j^{(l)}} = \frac{\partial \left(\sum_{n=0}^{m} \big(w_{kn}^{(l+1)} h_n^{(l)}\big) + b_k^{(l+1)} \right)}{\partial h_j^{(l)}} = w_{kj}^{(l+1)}, \tag{3.80}$$

and

$$\frac{\partial h_j^{(l)}}{\partial z_j^{(l)}} = \frac{d\sigma\big(z_j^{(l)}\big)}{dz_j^{(l)}} = h_j^{(l)}(1 - h_j^{(l)}). \tag{3.81}$$

Notice that in equation 3.81 it is assumed that the activation function is a sigmoid function.

Combining all these equations, one get the scalar expression for a single auxiliary variable. This is presented below along with the equivalent vector equation for the entire layer

$$\delta_j^{(l)} = \delta_j^{(l+1)} w_{kj}^{(l+1)} h_j^{(l)} (1 - h_j^{(l)}) \tag{3.82}$$

$$\boldsymbol{\delta}^{(l)} = \left(\big(\boldsymbol{W}^{(l+1)}\big)^T \boldsymbol{\delta}^{(l+1)} \right) \odot \boldsymbol{h}^{(l)} \odot \big(\mathbf{1} - \boldsymbol{h}^{(l)}\big). \tag{3.83}$$

Here $\odot$ denotes the Hadamard multiplication. This multiplicator uses element-wise product such that given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$

$$\big(\boldsymbol{a} \odot \boldsymbol{b}\big)_i = a_i \cdot b_i.$$

Notice that equation 3.83 assumes layers with the same number of neurons in them, however this can be adjusted. Now that the auxiliary variables $\boldsymbol{\delta}^{(l)}$ can be evaluated through $\boldsymbol{\delta}^{(l+1)}$ one have

to show that the auxiliary variables for the last layer are directly computable from the activations of that layer. Since there is only one output neuron the mathematics are the same as for the one neuron example earlier, see equations 3.74 and 3.75 for more details. However, it is necessary to convert this single value to an array. This can be done quite easily by just multiplying $\delta^{(L)}$ by an array of ones.

$$\boldsymbol{\delta}^{(L)} = \big(h^{(L)} - y_i\big)h^{(L)}(1 - h^{(L)}) \cdot \mathbf{1} \tag{3.84}$$

The next part is to evaluate the derivatives of the loss with respect to the weights and the biases in terms of the auxiliary variables

$$\frac{\partial \mathbb{L}}{\partial w_{jk}^{(l)}} = \frac{\partial \mathbb{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} \frac{\partial \Big(\sum_{k=0}^{m} \big(w_{jk}^{(l)} h_k^{(l-1)} \big) + b_j^{(l)} \Big)}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} h_k^{(l-1)}$$

$$\frac{\partial \mathbb{L}}{\partial b_j^{(l)}} = \frac{\partial \mathbb{L}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \frac{\partial \Big(\sum_{k=0}^{m} \big(w_{jk}^{(l)} h_k^{(l-1)} \big) + b_j^{(l)} \Big)}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{3.85}$$

In vector format these equations become

$$\nabla_{\boldsymbol{w}^{(l)}} \mathbb{L} = \boldsymbol{\delta}^{(l)} \big(\boldsymbol{h}^{(l-1)}\big)^T$$

$$\nabla_{\boldsymbol{b}^{(l)}} \mathbb{L} = \boldsymbol{\delta}^{(l)} \tag{3.86}$$

The idea for training now becomes the same as described in the one neuron per layer example.

3.4.5 L1 and L2 Regularization

L1 and L2 regularization are popular tools for preventing over-fitting and reducing complexity in neural networks. These methods accomplish this by adding a penalty term to the loss function, which penalizes the size of the weights and biases. This results in a new loss function. For L1, the new loss function becomes

$$\mathbb{L}(\boldsymbol{w}, \boldsymbol{b}) = \sum_{i=1}^{n} \Big(\mathbb{L}^{(i)} \left(y_i, \hat{y}_i\right) \Big) + \lambda \Big(||\boldsymbol{w}|| + ||\boldsymbol{b}|| \Big), \tag{3.87}$$

And for L2, the new loss function becomes

$$\mathbb{L}(\boldsymbol{w}, \boldsymbol{b}) = \sum_{i=1}^{n} \Big(\mathbb{L}^{(i)} \left(y_i, \hat{y}_i\right) \Big) + \lambda \Big(||\boldsymbol{w}||^2 + ||\boldsymbol{b}||^2 \Big), \tag{3.88}$$

In both these loss functions λ is a tuneable hyperparameter, and the larger λ is, the more complexity is penalized. $\mathbb{L}^{(i)}$ denotes the old loss function.

The key difference between L1 and L2 regularization is that L1 regularization tends to shrink many of the weights and biases to zero, hence creating sparse models. L2 regularization on the other hand tends to create models with low values, but not zero, for the weights and biases.

3.4.6 Scikit-learn

An introduction to Scikit-learn can be found in subsection 3.2.4, where the same library is used for logistic regression.

Scikit-learn also offers a powerful implementation of deep learning, which is both easy to implement, and flexible. In [12] one can find comprehensive information on Scikit-learns implementation of deep learning on classification problems. This includes coding examples, package details and hyperparameter introductions.

3.5 Hyperparameter tuning

When creating a machine learning model, one of the critical decisions to make is how to define the model's architecture. The model's architecture plays a crucial role in determining its performance, and selecting the optimal architecture can be a challenging task as it is often difficult to tell the optimal architecture for a given model right away.

To address this challenge, one approach is to explore a range of possibilities for the model architecture. This exploration for the optimal model architecture can be carried out either by a machine automatically, or manually by a person. Hyperparameters are parameters that remain constant throughout the training process, and they determine the model architecture. The process of exploring various possibilities of hyperparameters to find the optimal model architecture is known as hyperparameter tuning. Examples of hyperparameters include the number of trees to grow, the learning rate, and the fraction of columns to use for training, among other possibilities.

It is often unclear how hyperparameters affect the performance of machine learning algorithms, as the relationship between hyperparameters and model performance is often complex and non-linear. As a result, the performance of a model with a specific set of hyperparameters cannot be determined until the model is first trained and then tested. There are several methods for conducting this training and testing.

Grid search is a widely used hyperparameter tuning technique in machine learning that is both intuitive and easy to implement. The idea behind grid search is to try out every possible combination of hyperparameters within a pre-defined area of interest. Consequently, the approach for conducting grid search involves systematically searching over a predefined hyperparameter space to find the optimal combination of hyperparameters that results in the best model performance.

The grid search algorithm creates a grid of all possible combinations of hyperparameters based on a set of predefined values for each hyperparameter. The model is trained and evaluated for each combination of hyperparameters in the grid, and the set of hyperparameters that produces the best model performance is selected as the optimal combination.

In the real world, training a model often requires a considerable amount of time, and grid search is a computationally intensive method. Especially when the number of hyperparameters and their respectable domains are large. The curse of dimensionality afflicts grid search because the quantity of models that must be trained increases exponentially with the number of hyperparameters involved. However, grid search is a simple and effective technique for finding the optimal combination of hyperparameters when the hyperparameter space is relatively small.

Random search is another popular hyperparameter tuning technique used in machine learning. Unlike grid search, which searches through a predefined set of hyperparameters in a grid-like manner, random search samples hyperparameters randomly from a defined hyperparameter space.

The idea behind random search is that, in many cases, only a few hyperparameters have a significant impact on model performance. By randomly sampling from a wide range of possible hyperparameters, the technique allows for a greater exploration of the hyperparameter space. As a consequence, random search is more likely to identify the optimal values of the most influential hyperparameters.

When the hyperparameters impact on the model performance varies significantly, the model is said to have a lower effective dimensionality. Figure 3.13 illustrates the contrast between grid search and random search in terms of their ability to deal with low effective dimensionality. In this given example, the model $F(x, y)$ is dependent on two variables - x and y. The impact of x on the model F is significant and is represented by a subspace on the top of each plot, whereas y has a relatively minor impact and is represented by a subspace on the left of each plot. Thus, it can be approximated that $F(x, y) \approx F(x)$. In the grid search layout, the points are uniformly distributed in the original space, but not efficiently distributed in the subspace. On the other hand, the random search layout shows slightly less uniform distribution of points in the original space, but a more efficient distribution in the subspace. As a result, in this example, random search is more likely to find hyperparameters that will produce a better model than grid search.

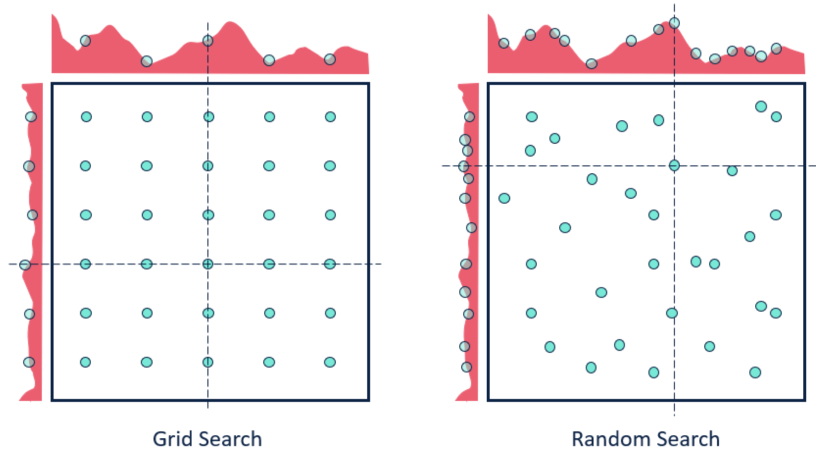

Figure 3.13: An illustration showing how a random search can be more beneficial than a grid search when some hyperparameters have a small influence on the model performance. Taken from [20]

While grid search will not be utilized in this thesis for hyperparameter tuning, it is important to note that this technique offers a systematic and structured approach to exploring hyperparameters by exhaustively searching over a predefined range of values. The inclusion of grid search in the theoretical framework aims to provide an understanding of how each possible combination of hyperparameters can influence the model, which is the ultimate goal of hyperparameter tuning. However, due to the curse of dimensionality, it is not a practical approach to do grid search in this thesis.

In this thesis, random search will be employed to initialize Bayesian optimization for hyperparameter tuning. The inclusion of random search in the theoretical framework serves another purpose as well, as it provides a clear and intuitive way to demonstrate that, despite initially seeming inferior to grid search, random search can actually outperform it in many scenarios.

3.5.1 Bayesian optimization

Grid search and random search are popular techniques for hyperparameter tuning in modern machine learning research [21]. However, these methods leave out something potentially very useful. Both have a limitation in that they do not leverage any information gained from previous iterations to identify optimal hyperparameter values. They simply iterate through a vast number of possible hyperparameter combinations without regard to the outcomes of previous searches. This is where Bayesian optimization offers a significant advantage.

By contrast, Bayesian optimization incorporates previous iterations into its search process, making

it an efficient and powerful hyperparameter tuning technique. By iteratively updating a probabilistic model of the objective function, Bayesian optimization is able to quickly identify promising areas of the hyperparameter space and focus its search there. This results in faster convergence and more accurate identification of optimal hyperparameter values, making Bayesian optimization an ideal technique for tuning complex models with a large number of hyperparameters.

There should be noted that there also exists other ways to tune hyperparameters where one take into account the results of previous iterations. Design of Experiments (DOE), is such a way [32]. DOE is done by hand, such that the next hyperparameter values that will be tested are decided by a person, and then put into the computer manually. DOE involves selecting hyperparameters, determining their values, and deciding on the number of experiments to conduct. The goal is to understand how the response is affected by the settings of the hyperparameters involved. DOE for hyperparameter tuning enables efficient exploration and optimization, however it lacks the automation that Bayesian optimization has to offer. Research has also shown that Bayesian optimization can outperform manual hyperparameter tuning performed by human experts [21].

In Bayesian optimization, the objective is to identify the maximum value of an unknown function F at a given sampling point $\boldsymbol{x}$,

$$\boldsymbol{x}^* = \underset{\boldsymbol{x} \in A}{\operatorname{argmax}} F(\boldsymbol{x}). \tag{3.89}$$

The search space for the hyperparameters $\boldsymbol{x}$ is denoted as A.

Bayesian optimization is based on Bayes' theorem [11], which states that the posterior probability $P(M|E)$ of a model M, given evidence data E, is proportional to the likelihood $P(E|M)$ of observing E given model M, multiplied by the prior probability $P(M)$:

$$P(M|E) \propto P(E|M)P(M) \tag{3.90}$$

The fundamental concept of Bayesian optimization is reflected by this formula. At the heart of Bayesian optimization lies the principle of combining the prior distribution of the objective function $F(\boldsymbol{x})$ with the sampled information to obtain the posterior. The ultimate goal is to locate the maximum value of $F(\boldsymbol{x})$ using an utility function a, also known as the acquisition function. The prior distribution is updated as new data is collected, and the acquisition function utilizes this updated information to determine the next point to be evaluated. This iterative process is repeated until the maximum number of iterations is reached, or some other stopping criterion is met.

As the prior distribution for Bayesian optimization a Gaussian process is often preferred. A Gaussian process is a stochastic process in which any random variable has a multivariate Gaussian distribution, making the Gaussian process a generalization of the Gaussian probability distribution. In the case of D hyperparameters, the hyperparameters are represented by $\boldsymbol{x} \in \mathbb{R}^D$. The Gaussian process is defined by a mean function $\mu : \mathbb{R}^D \to \mathbb{R}$ and a covariance function $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$, such that the Gaussian process can be denoted as:

$$F(\boldsymbol{x}) \sim \mathrm{GM}(\mu(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}')). \tag{3.91}$$

Let

$$\boldsymbol{x}_{1:t} = [\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_t]^T \quad \text{and} \quad \boldsymbol{F}_{1:t} = [F(\boldsymbol{x}_1), F(\boldsymbol{x}_2), \ldots, F(\boldsymbol{x}_t)]^T.$$

Now, for a given set of t observations $\{\boldsymbol{x}_i, F(\boldsymbol{x}_i)\}_{i=1}^t$, the probability of any finite set of $\boldsymbol{F}_{1:t}$ is now assumed to be Gaussian,

$$\boldsymbol{F}_{1:t} \sim N(\boldsymbol{\mu}(\boldsymbol{x}_{1:t}), \boldsymbol{K}(\boldsymbol{x}_{1:t}, \boldsymbol{x}_{1:t})). \tag{3.92}$$

Where

$$\boldsymbol{K}(\boldsymbol{x}_{1:t}, \boldsymbol{x}_{1:t}) = \begin{bmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & k(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_t) \\ k(\boldsymbol{x}_2, \boldsymbol{x}_1) & k(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & k(\boldsymbol{x}_2, \boldsymbol{x}_t) \\ \vdots & \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_t, \boldsymbol{x}_1) & k(\boldsymbol{x}_t, \boldsymbol{x}_2) & \dots & k(\boldsymbol{x}_t, \boldsymbol{x}_t) \end{bmatrix} \tag{3.93}$$

Several options exist for the correlation function k, but a popular and simple choice is

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\frac{1}{2}||\boldsymbol{x}_i - \boldsymbol{x}_j||^2). \tag{3.94}$$

The anticipated distribution of the Gaussian process is now analytically feasible. If there is a new point $\boldsymbol{x}_{t+1}$, the joint probability distribution of the established values $\boldsymbol{F}_{1:t}$ and the predicted function $F(\boldsymbol{x}_{t+1})$ is expressed as follows

$$\begin{bmatrix} \boldsymbol{F}_{1:t} \\ F(\boldsymbol{x}_{t+1}) \end{bmatrix} \sim N\left(\begin{bmatrix} \boldsymbol{\mu}(\boldsymbol{x}_{1:t}) \\ \mu(\boldsymbol{x}_{t+1}) \end{bmatrix}, \begin{bmatrix} \boldsymbol{K}(\boldsymbol{x}_{1:t}, \boldsymbol{x}_{1:t}) & \boldsymbol{k} \\ \boldsymbol{k}^T & k(\boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+1}) \end{bmatrix} \right), \tag{3.95}$$

where

$$\boldsymbol{k} = [k(\boldsymbol{x}_{t+1}, \boldsymbol{x}_1), k(\boldsymbol{x}_{t+1}, \boldsymbol{x}_2), ..., k(\boldsymbol{x}_{t+1}, \boldsymbol{x}_t)]^T.$$

By utilizing $\boldsymbol{\mu}(\boldsymbol{x}_{1:t}) = 0$, the problem can now be simplified. As a result, the predictive posterior distribution of $F(\boldsymbol{x}_{t+1})$ becomes Gaussian.

$$F(\boldsymbol{x}_{t+1})|\boldsymbol{F}_{1:t} \sim N(\mu_{t+1}(\boldsymbol{x}_{t+1}|\boldsymbol{F}_{1:t}), \sigma^2_{t+1}(\boldsymbol{x}_{t+1}|\boldsymbol{F}_{1:t})), \tag{3.96}$$

where

$$\mu_{t+1}(\boldsymbol{x}_{t+1}) = \boldsymbol{k}^T \boldsymbol{K} \boldsymbol{F}_{1:t},$$

and

$$\sigma^2_{t+1}(\boldsymbol{x}_{t+1}) = k(\boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+1}) - \boldsymbol{k}^T \boldsymbol{K}^{-1} \boldsymbol{k}.$$

After obtaining the posterior distribution of the objective function, Bayesian optimization employs the acquisition function a to determine the maximum of the function F,

$$\boldsymbol{x}_{next} = \operatorname*{argmax}_{\boldsymbol{x}} a(\boldsymbol{x}|\boldsymbol{x}_{1:t}, F_{1:t}). \tag{3.97}$$

It is typically assumed that a high value in the acquisition function a corresponds to a high value of the objective function F. Consequently, maximizing the acquisition function is equivalent to maximizing the function F,

$$\boldsymbol{x}^+ = \operatorname*{argmax}_{\boldsymbol{x} \in \boldsymbol{x}_{1:t}} F(\boldsymbol{x}). \tag{3.98}$$

Here, the hyperparameters that yield the best value obtained so far are denoted as $\boldsymbol{x}^+$. In other words, the acquisition function is used to explore the hyperparameter space in order to find the next set of hyperparameters to evaluate.

Probability of Improvement

Historically, the first proposed acquisition function was Probability of Improvement [25]. It aims to maximize the probability of improvement over the current best value. This function calculates the probability that the next point will be better than the current best point,

$$a_{PI}(\boldsymbol{x}) = P(F(\boldsymbol{x}) > F(\boldsymbol{x}^+)). \tag{3.99}$$

$F(\boldsymbol{x})$ is Gaussian, and $a_{PI}(\boldsymbol{x})$ can thus be expressed as

$$\begin{aligned} a_{PI}(\boldsymbol{x}) = 1 - P(F(\boldsymbol{x}) \leq F(\boldsymbol{x}^+)) &= 1 - P\left(\frac{F(\boldsymbol{x}) - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})} \leq \frac{F(\boldsymbol{x}^+) - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})}\right) \\ &= 1 - \Phi\left(\frac{F(\boldsymbol{x}^+) - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})}\right) = \Phi\left(\frac{\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+)}{\sigma(\boldsymbol{x})}\right). \end{aligned} \tag{3.100}$$

One limitation with using Probability of Improvement is that the algorithm shares similarities with the gradient descent algorithm. Consequently, the next sampling point may be confined to a restricted range, leading to the potential of the optimization algorithm becoming trapped in a local optimum.

To address this issue, a possible solution is to introduce an explanatory parameter ξ into the function,

$$a_{PI}(\boldsymbol{x}) = P(F(\boldsymbol{x}) \geq F(\boldsymbol{x}^+) + \xi) = \Phi\left(\frac{\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+) - \xi}{\sigma(\boldsymbol{x})}\right). \tag{3.101}$$

The new sampling point will replace the current optimal value only if the difference between the value of the subsequent sampling point and the current optimal value exceeds ξ. Increasing the value of ξ will encourage exploration, enabling the algorithm to seek additional solutions beyond the current optimum.

The Probability of Improvement function has been used in a variety of applications, such as optimization of expensive black-box functions and reinforcement learning. It is also computationally efficient and has a simple analytical form, making it an attractive choice for practical applications.

In general, even though the Probability of Improvement function is able to balance exploration and exploitation, Probability of Improvement tends to favor exploitation over exploration, as it focuses on finding points that are likely to improve the current best value.

Expected Improvement

Expected Improvement is another popular acquisition function. I differs from Probability of Improvement by taking into account not just the probability of improvement, but also the extent of improvement. The Expected Improvement function measures the improvement in the objective function that can be expected by evaluating a new point in the search space. It does this by comparing the predicted value of the objective function at the new point with the current best value found so far. This is achieved through the use of an improvement function, denoted as $I(\boldsymbol{x})$,

$$I(\boldsymbol{x}) = \max\{F(\boldsymbol{x}) - F(\boldsymbol{x}^+), 0\}. \tag{3.102}$$

To clarify, if the predicted value is greater than the current best known value, the improvement function $I(\boldsymbol{x})$ takes on a positive value. However, if the predicted value is not better than the current best known value, then $I(\boldsymbol{x})$ is assigned a value of zero.

In order to compute the expected improvement, a reparameterization technique is employed. Recall that the distribution of $F(\boldsymbol{x})$ follows a normal distribution with a mean of $\mu(\boldsymbol{x})$ and a variance of $\sigma^2(\boldsymbol{x})$. To facilitate the computation, a new variable z is introduced, which is also normally distributed with a mean of zero and a variance of one; $z \sim N(0,1)$. By using this new variable, $F(\boldsymbol{x})$ can be expressed as $F(\boldsymbol{x}) = \mu(\boldsymbol{x}) + \sigma(\boldsymbol{x})z$, which is still normally distributed with a mean of $\mu(\boldsymbol{x})$ and a variance of $\sigma^2(\boldsymbol{x})$. In this reparameterization, the improvement function $I(\boldsymbol{x})$ can be rewritten as

$$I(\boldsymbol{x}) = \max\{F(\boldsymbol{x}) - F(\boldsymbol{x}^+), 0\} = \max\{\mu(\boldsymbol{x}) + \sigma(\boldsymbol{x})z - F(\boldsymbol{x}^+), 0\} \tag{3.103}$$

As a result of the reparameterization, the expectation of the improvement can be expressed as

$$\mathrm{EI}(\boldsymbol{x}) \equiv \mathbb{E}[I(\boldsymbol{x})] = \int_{-\infty}^{\infty} I(\boldsymbol{x})\phi(z)dz = \int_{-\infty}^{\infty} \max\{F(\boldsymbol{x}) - F(\boldsymbol{x}^+), 0\}\phi(z)dz \tag{3.104}$$

Here, $\phi(z)$ represents a standardized normally distributed variable z, with a probability density

$$\phi(z) = \frac{1}{\sqrt{2\pi}}\exp(-\frac{1}{2}z^2).$$

To calculate this integral, it is necessary to eliminate the *max* operator. This can be achieved by splitting the integral into two parts based on whether $F(\boldsymbol{x}) - F(\boldsymbol{x}^+)$ is positive or negative. The point at which this transition occurs, where $F(\boldsymbol{x}) = F(\boldsymbol{x}^+)$, is labeled as z_0.

$$F(\boldsymbol{x}) = F(\boldsymbol{x}^+) \Rightarrow \mu(\boldsymbol{x}) + \sigma(\boldsymbol{x})z = F(\boldsymbol{x}^+) \Rightarrow z = \frac{F(\boldsymbol{x}^+) - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})} = z_0$$

Consequently, the integral can be split in two,

$$EI(\boldsymbol{x}) = \underbrace{\int_{-\infty}^{z_0} I(\boldsymbol{x})\phi(z)dz}_{\text{Zero since } I(\boldsymbol{x})=0} + \int_{z_0}^{\infty} I(\boldsymbol{x})\phi(z)dz.$$

Now, the expected improvement can be computed as follows:

$$\begin{aligned}
EI(\boldsymbol{x}) &= \int_{z_0}^{\infty} \max\{F(\boldsymbol{x}) - F(\boldsymbol{x}^+), 0\}\phi(z)dz = \int_{z_0}^{\infty} (\mu(\boldsymbol{x}) + \sigma(\boldsymbol{x})z - F(\boldsymbol{x}^+)\phi(z))dz \\
&= \int_{z_0}^{\infty} (\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+)\phi(z))dz + \int_{z_0}^{\infty} \sigma(\boldsymbol{x})z\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}z^2}dz \\
&= (\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+))\underbrace{\int_{z_0}^{\infty} \phi(z)dz}_{1-\Phi(z_0)} + \sigma(\boldsymbol{x})\int_{z_0}^{\infty} \frac{1}{\sqrt{2\pi}}ze^{-\frac{1}{2}z^2}dz \\
&= (\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+))\underbrace{(1-\Phi(z_0))}_{\Phi(-z_0)} + \sigma(\boldsymbol{x})\underbrace{\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}z_0^2}}_{\phi(-z_0)} \\
&= (\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+))\Phi(-z_0) + \sigma(\boldsymbol{x})\phi(-z_0)
\end{aligned} \tag{3.105}$$

As a result, the acquisition function for the Expected Improvement can now be evaluated,

$$a_{EI}(x) = (\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+))\Phi\left(\frac{\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+)}{\sigma(\boldsymbol{x})}\right) + \sigma(\boldsymbol{x})\phi\left(\frac{\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+)}{\sigma(\boldsymbol{x})}\right). \tag{3.106}$$

Here $\Phi(.)$ is a cumulative distribution function.

Similar to Probability of Improvement, an explanatory parameter ξ can be incorporated to regulate exploration.

$$a_{EI}(\boldsymbol{x}) = (\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+))\Phi\left(\frac{\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+) - \xi}{\sigma(x)}\right) + \sigma(\boldsymbol{x})\phi\left(\frac{\mu(\boldsymbol{x}) - F(\boldsymbol{x}^+) - \xi}{\sigma(\boldsymbol{x})}\right). \quad (3.107)$$

Similar to previous, a larger ξ will encourage exploration.

The Expected Improvement function has two main advantages. Firstly, it is able to balance exploration and exploitation by placing a higher emphasis on unexplored areas of the search space where the improvement could potentially be large. Secondly, it has a straightforward analytical form that allows for efficient optimization.

The Expected Improvement function has been widely used in various applications, such as optimization of expensive black-box functions, and experimental design. Its versatility, efficiency, and ability to balance exploration and exploitation make it a popular choice in Bayesian optimization.

Confidence Bound

The Gaussian Process Upper Confidence Bound (GP-UCB), often referred to as Confidence Bound, is another popular acquisition function used in Bayesian optimization [19]. The Confidence Bound function is based on the upper confidence bound of the posterior distribution of the objective function, which represents the uncertainty of the model's prediction.

The Confidence Bound function aims to balance exploration and exploitation by selecting points in the search space that have high potential for improvement but are also uncertain. Specifically, the Confidence Bound function selects the point with the highest upper confidence bound, which corresponds to the point with the highest potential for improvement while still having a high level of uncertainty. The acquisition function thus takes the form,

$$a_{GP-UCB}(\boldsymbol{x}) = \mu(\boldsymbol{x}) + \kappa\sigma(\boldsymbol{x}). \quad (3.108)$$

In this context, the parameter κ is adjustable and serves to balance the degree of exploitation versus exploration. A larger value of κ corresponds to a greater emphasis on exploration. It is not uncommon for more sophisticated algorithms to update the value of κ based on the number of iterations completed within the algorithm. An example of this is showed below,

$$\kappa_t = \frac{1}{\sqrt{\upsilon t}}. \quad (3.109)$$

Here υ is a constant, an t denotes the number of iterations performed. This approach is often utilized to promote exploration at the beginning of the process, while shifting towards exploitation as the process continues.

One advantage of the Confidence Bound function is that it provides a measure of the trade-off between exploration and exploitation in a principled manner. This can be useful in scenarios where the cost of evaluating the objective function is high, and selecting points with high potential for improvement is critical.

Figure 3.14 illustrates examples of different acquisition functions and their settings. It can be observed that the location of the maximum value for each of the three functions varies, but not as much as the location of the maximum value varies with respect to the settings of the exploration parameters. This suggests that the selection of exploration parameters may be more influential in determining the behavior of the acquisition function than the choice of the acquisition function itself.

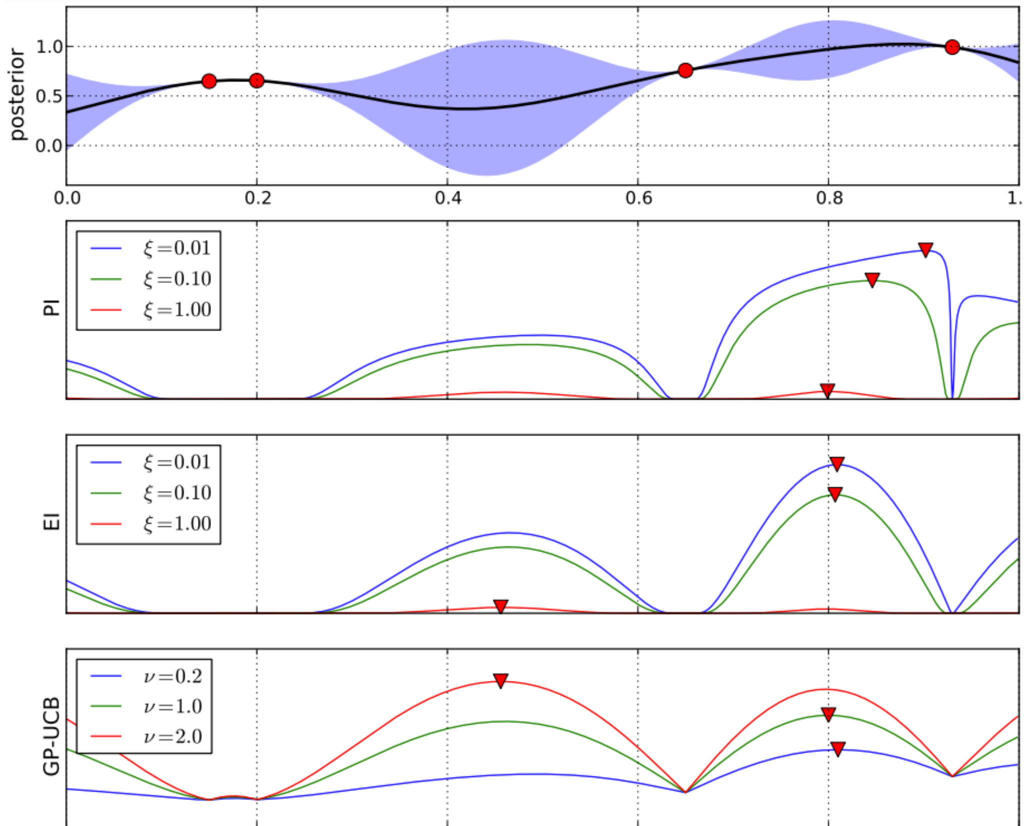

Figure 3.14: The figure presents examples of acquisition functions and their settings. The top panel displays the Gaussian process posterior, while the other panels show the corresponding acquisition functions, which are probability of improvement, expected improvement, and upper confidence bound, respectively. A triangle marker indicates the location of the maximum value for each function. Taken from [19]

.

3.5.2 Optuna

Optuna is a widely used open source automatic optimization framework for hyperparameters. It is designed to automate the process of hyperparameter tuning by efficiently searching for the best hyperparameter values through trial and error. One of the main methods used by Optuna for hyperparameter tuning is Bayesian optimization, which is based on the principles described in this sub-chapter. However, the methods used in Optuna are highly optimized to improve efficiency and effectiveness. In this thesis, all hyperparameter tuning is performed using Bayesian optimization implemented in Optuna. For further information about Optuna, see [24].

Chapter 4

Data Preparation and Visualization

In this chapter, a more thorough understanding of the data provided by Sparebank1 will be offered. The chapter will begin with an introduction to the different datasets provided by Sparebank1, and then proceed to show some visualizations. In the end, the methods used to prepare the data for analysis will be presented.

4.1 The different datasets

The datasets used in this thesis were provided by Sparebank1. There were originally given four datasets, each dataset containing different information regarding a customer. All the datasets will get a brief introduction below. Table 4.1 provides a summary of all the datasets. A comprehensive list of all the variables in all the original datasets along with their respective explanations are given in Appendix A.

Table 4.1: A summary of all the different datasets.

Datasets	Description	Number of variables	Number of observations
Fundamental	Basic information regarding customers and their credit cards	12	262 773
Appliance	Information regarding customers, this includes income, children and marital status	23	20 933
Historical credit card	Information regarding customers credit card history	23	174 492
Historical transactions	Information regarding customers transactions to their credit card account	12	5 475
Final dataset	The final dataset used for prediction after pre-prosessing	1199	262 773

The fundamental dataset consists of 262773 observations and 12 variables, including the response variable "AktivEtterPassiv". The other variables in the dataset contain information regarding the customer, such as what sex the customer are, the age of the customer, the first and last time the customers used their credit card, and so on. Each observation in the fundamental dataset represents a single customer at a given date. The dates range from May 2020 all the way to February 2023.

The information in the appliance dataset is collected from when a customer applied for the credit card, which means that there is only one observation per customer. The dataset consists of 20933 observations and 23 variables.

The data in the appliance dataset contain information that was useful for the bank when determining if the customer should get a credit card, and the balance on the credit card. As a result, the variables in the dataset contain information regarding the customers appliance, such as what balance the customer wants on their credit card, what balance the customer got on their credit card, the gross income of the customer, different types of loans the customer might have, if the customer has children, and in that case how many, and so on.

The information from the historical credit card dataset is collected from when a customer has used their credit card for payment. Each observation in this dataset represents a single customer at a given period. The historical credit card dataset consists of 174492 observations and 23 variables, where the variables contains the amount of money that each customer has spent with their credit card on various products within a specified timeframe.

The various products are put into different groups, like groceries, transport and restaurants. The different timeframes are 12 and 3 months. As a result, a given variable can consist of the amount of money spent on groceries for the last 12 months.

The information from the historical transactions dataset is collected from when a customer has made transactions from or to a known bank account that the customer possesses. Each observation in this dataset represents a single customer at a given period. The historical transactions dataset consists of 5475 observations and 12 variables, where the variables contain the amount of money transferred or the amount of transactions made within a 12 months period.

The final dataset is the dataset after preprosessing, and is the dataset used for prediction.

4.2 Visualization

Visualizing data can be highly beneficial in gaining a more profound understanding of how the data interact.

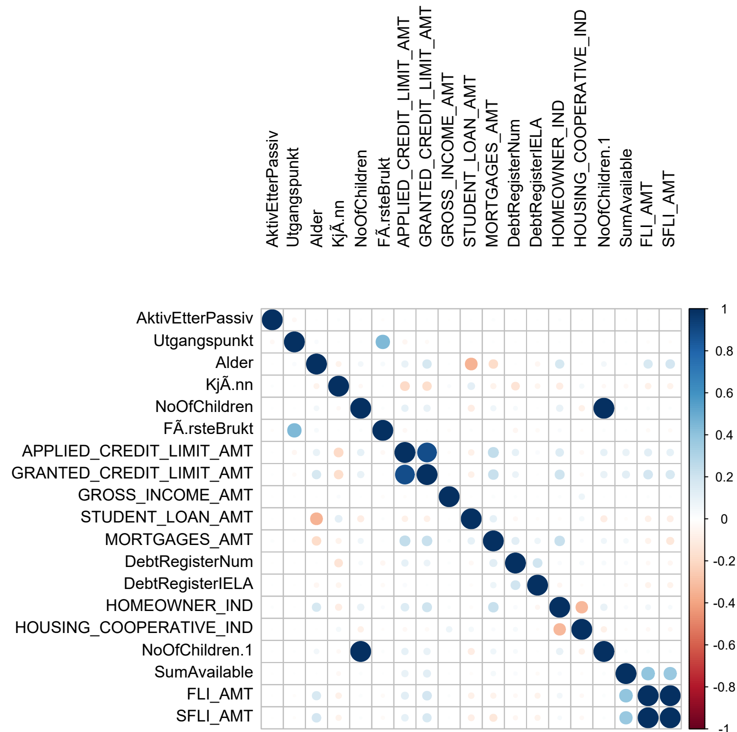

Figure 4.1: Correlations between different variables within the merged dataset prior to preprocessing

Before beginning the visualization, the fundamental dataset and the appliance dataset were merged by using the variable "BK_ACCOUNT_ID", which is the personal account number of a customer. This resulted in one dataset that consists of 262 773 observations and 34 variables, which is more or less the same dataset as the fundamental dataset, just with more explanatory variables.

Exploring the relationships between variables can provide valuable insight. Figure 4.1 visualizes the correlations between different variables. To calculate correlations, categorical variables had to be excluded.

Observe that "APPLIED_CREDIT LIMIT_AMT" and "GRANTED_CREDIT_LIMIT_AMT" are strongly correlated. This could be interpreted as that the majority of customers are granted a credit card limit close to their requested amount. One can also observe that "FLI_AMT" and "SFLI_AMT" are highly correlated. "FLI_AMT" is a number that tells how well a person can manage a loan, and "SFLI_AMT" is a stress-test where the loans interests are raised with 5 percentage point. Together with "FLI_AMT" and "SFLI_AMT", "SumAvailable" also seems to be relatively correlated. Notice that "NoOfChildren" and "NoOfChildren.1" also is really correlated, these two variables are the same variables, but one of them was from the appliance dataset, whereas the other was from the fundamental dataset. This is a duplication that was removed later.

A correlation plot done with all the variables on the merged dataset after the pre-processing can be found in Appendix B.

Figure 4.2 displays density plots of selected variables. This is a visualization tool that presents the distributions of continuous variables. In this case, it is used to compare the distribution of the same explanatory variable for two different responses in the response variable "AktivEtterPassiv". The colors red and blue represent the responses 0 (remaining passive) and 1 (becomes active), respectively. The varying densities depicted in the plots indicate differences between customers who remain passive and customers who become active, providing insights as to how some of the explanatory variables may help to predict the response variable.

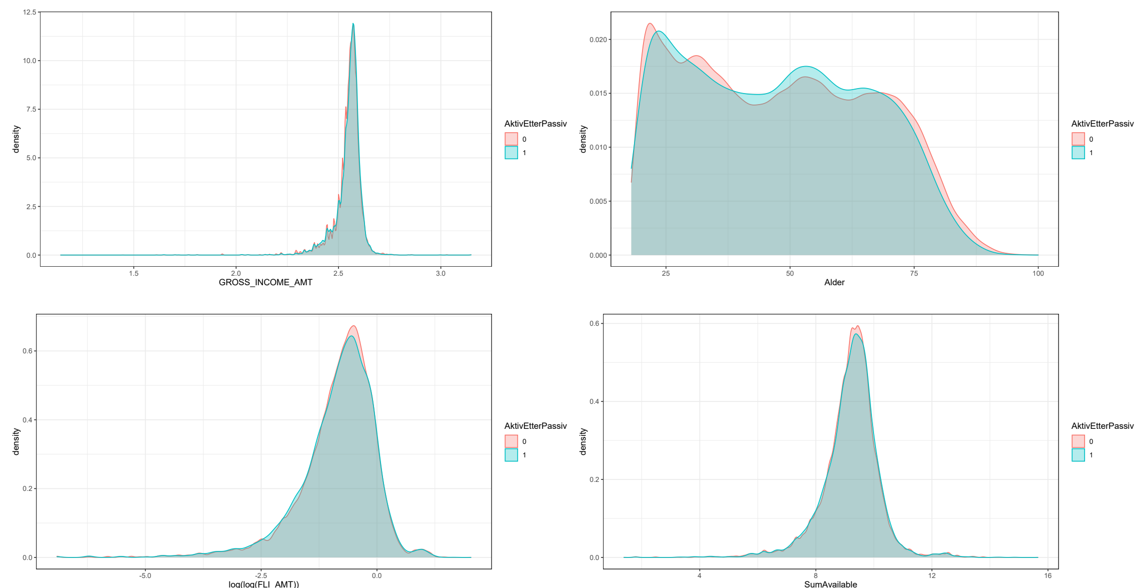

Figure 4.2: Density plots of selected explanatory variables. Red shows the densities for observations that remain passive, while blue shows the densities for observations that become active. The figures follow this layout: Top left: ("GROSS_INCOME_AMT") The income the customer has. Top Right: ("Alder") The age of the customer. Bottom left: (log(log("FLI_AMT"))) The natural logarithm of the natural logarithm of a liquidity indicator made by Sparebank1 on how well the customer can handle a loan. Bottom right: ("SumAvailable") The amount of money available to a customer.

Looking at the densities for the different explanatory variables it becomes clear that there is not much of a difference between the densities of the observations becoming active and the observations

who remain passive. However, although subtle, there are some differences.

If one were to look at "Alder", one can notice a trend; older customers tend to be slightly less likely to become active, and the same seems to be the case for the youngest customers. There is also a very subtle trend in the variable "FLI_AMT", where a lower value here will increase the chance of becoming active. Although the other two plots exhibit some variation in densities between the two groups, they are not as distinct as in the other plots.

Let us also look at the historical datasets, these datasets contain a huge amount of observations for each feature. Most of these observations are zero, as there are many categories in which the customer has not used their credit card.

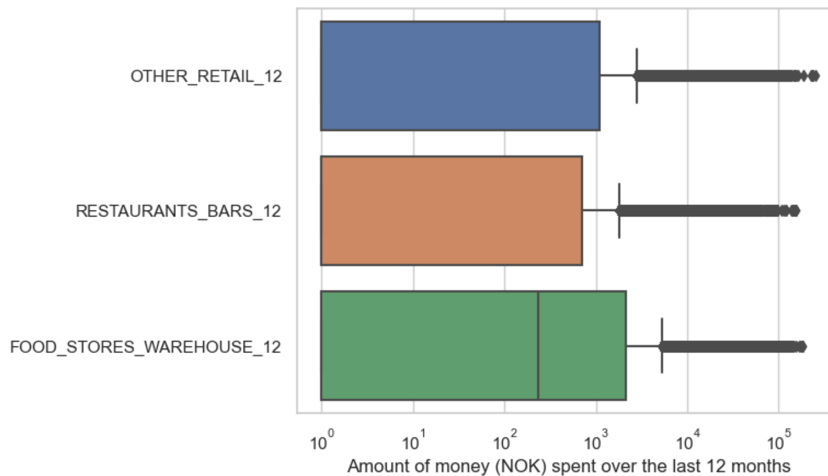

Figure 4.3: The figure shows three box plots on how much money is spent in different categories for the last 12 months.

In Figure 4.3 box plots of how much money is spent in selected categories is shown. Most customers use less than 10 000 NOK in each category, however, there are some outliers that use substantially more.

The line right between 100 and 1000 NOK in the category "FOOD_STORES_WAREHOUSE_12" shows the median amount of money spent. In the two other categories, this median is zero.

4.3 Pre-Processing

Data pre-processing refers to the manipulation or removal of data before its usage to ensure or improve the predictive models performance.

4.3.1 The first two datasets

Before any pre-processing is done, some of the explanatory variables had to removed from the dataset made by merging the fundamental dataset and the appliance dataset. This removal had to be done due to the fact that many of the explanatory variables contained information regarding if the customers had become active, hence ruining the whole predicting task.

There are 4 of these variables, "SisteKortbruk", "SisteTransaksjon", "Revolver" and "Fullpayer". The first two explanatory variables will contain a date after the one month predicting interval if a person has used their credit card or made any transactions after that month, hence telling if a person is going from passive to active. While the last two explanatory variables won't entirely hinder the prediction task, they do indicate whether a customer has used their credit card. Passive

users cannot be classified as full-payers or revolvers, so the variables representing these categories will have a value of zero. However, if these variables change to one, it will indicate that the customer has used their credit card again.

After removing some of the explanatory variables, the next part is making the data available for the computer. This is only done for the variables ”ForsteBrukt” and ”Utgangspunkt”. This is because both these variables display dates.

For ”ForsteBrukt” the values were changed to days between the date that is displayed, and the date at ”Utgangspunkt”. This is done because it is believed that the predicting models can make better use of that type of data, as it now tells directly how many days it is between the first time a customer uses their credit card and the predicting date.

For ”Utgangspunkt”, two new explanatory variables were created. The two new variables being created are ”Utgangspunkt_år” and ”Utgangspunkt_måned”, containing information of what year and what month it is.

Afterwards, the pre-processing continued with feature construction. Two new variables were created. One by taking the difference between ”APPLIED_CREDIT_LIMIT_AMT” and ”GRANTED_CREDIT_LIMIT_AMT”, and the other one by dividing ”APPLIED_CREDIT_LIMIT _AMT” by ”GRANTED_CREDIT_LIMIT_AMT”. Both of these feature constructions were carried out to capture any potential reason behind why a customer may not be granted the credit limit they requested.

The variable ”CREATED_DT” tells when the appliance for the credit card was made. Along with the date, this variable also contains the hour of the day, hence making it possible to create a new feature with that information. There were also made a new feature which is the difference between the variable ”FørsteBrukt” and the date at ”CREATED_DT”. This new feature contains the information of how many days there are between the appliance of the credit card and the actual usage of the credit card.

Dummy encoding

Handling categorical values with neural networks can often be quite challenging. A common approach to deal with this issue is to use something called dummy encoding. Dummy encoding, also known as one-hot encoding, is a popular technique used in machine learning and data analysis to convert categorical variables into a numerical representation that can be used by predicting models.

Dummy encoding creates binary variables for each unique category within a categorical feature. Each binary variable represents whether a specific category is present or not. If a category is present for a particular observation, the corresponding binary variable will be assigned a value of 1, and if not, it will be assigned a value of 0.

In the merged dataset there were many missing values. A complete list of all the explanatory variables with explanation for each of the different datasets, as well as how many observations that were missing for each variable, is included in Appendix A. As the reasons for the missing values were unknown, they could potentially hold valuable information for the computer. Therefore, it was crucial to preserve the information about missing values in the data after pre-processing. The significance of this is demonstrated in Figure 4.4, which highlights the substantial decrease in the probability of becoming an active user due to missing values in the data.

After performing dummy encoding for both categorical and missing values, there were still some ”not a number” (NaN) values in the numerical variables. To handle this, these NaN values were replaced with either the median or the mean of the remaining data in that variable.

The new dataset that was created from the fundamental dataset and the appliance dataset now consisted of 103 variables and 262 773 observations.

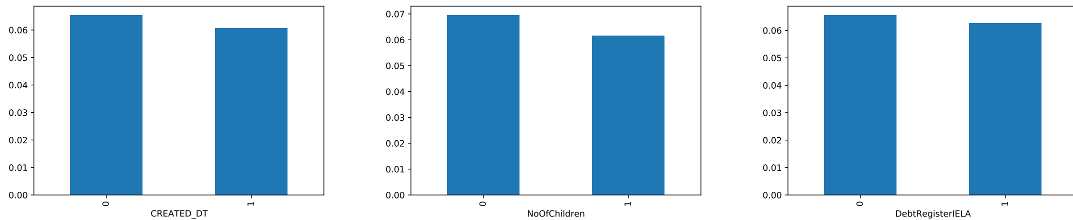

Figure 4.4: The three figures show the importance of keeping information about missing values. Each plot shows the proportion of data with a positive outcome of the binary response variable "AktivEtterPassiv" on the y-axis. The x-axis represents the presence or absence of missing observations, where 0 indicates the absence of missing observations and 1 indicates their presence.

4.3.2 The last two datasets

The historical credit card dataset, and the historical transactions dataset were on a slightly different format than desired. This led to the necessity to transform these datasets in such a way that they could be merged with the fundamental dataset. For doing that, it was needed to put every bit of information regarding a customer into one observation. This is illustrated in Figure 4.5.

BK_ACCOUNT_ID	PeriodId	FOOD_STORE	OTHER_TRANSPORT
1	2020.01	2000	500
1	2020.02	3000	0
1	2020.03	2500	3000
2	2020.01	6000	1000
2	2020.02	8000	2000
3	2020.02	500	0
3	2020.03	300	0

BK_ACCOUNT_ID	FOOD_STORE _2020_01	FOOD_STORE _2020_02	FOOD_STORE _2020_03	OTHER _TRANSPORT _2020_01	OTHER _TRANSPORT _2020_02	OTHER_ TRANSPORT _2020_03
1	2000	3000	2500	500	0	3000
2	6000	8000	0	1000	2000	0
3	0	500	300	0	0	0

Figure 4.5: An illustration made to show how the historical datasets has been changed to later be merged with the other datasets. This is a big simplification, and the upper table represents the data given by Sparebank1, and the lower table represents the transformed table that later would be merged with the other datasets.

In the datasets provided by Sparebank1 the periods ranged all the way from 2019 to 2023, consequently the transformed datasets contained a significant number of explanatory variables.

The historical credit card dataset after transformation contained 820 explanatory variables and 28 375 observations. While the historical transactions dataset after transformation contained 281 explanatory variables and 5081 observations.

Every dataset could then be combined using the variable "BK_ACCOUNT_ID". This led to some new NaN values because not all bank account id's had all the historical data. These NaN values were changed into zeros as it was believed that the reason why the NaN values existed was that there just had not been an event where the money had been spent or transferred.

Now that every dataset was combined, some modifications on some of the explanatory variables that came from the historical datasets had to be done. Some of the customers in the dataset became active, and then passive again. As a result, for a given customer at a given period of prediction, some of the explanatory variables contained information about future usage of the credit card. This

is obviously bad for the predicting task. Hence, for every observation, it was searched through all the explanatory variables to see if there was some information about the future, and if there were some, it was deleted.

In the end, `MinMaxScaler()` from `sklearn.preprocessing` in python was used as a way to scale all the explanatory variables in the whole dataset between zero and one. This is a standard procedure in machine learning, and is done as it often makes it easier for the predicting models to learn efficiently.

Chapter 5

Methods and Hyperparameters, an Overview

In this chapter, a brief summary of the hyperparameters of each of the predictive models along with the techniques used to construct and improve the predictive models will be provided. All models were developed using the programming language python. The code implemented are presented in Appendix G, while the underlying theory is explained in Chapter 2.

5.1 The predicting models

In this section, the predictive models along with their corresponding hyperparameters will be presented.

5.1.1 Logistic Regression

The modeling of logistic regression involves using the `LogisticRegression()` function imported from the `sklearn.linear_model` sub-library. This predicting model contains several hyperparameters, all of which can be found in [14]. For the purpose of this thesis, it has been selected five key hyperparameters for tuning, which are listed in Table 5.1 along with their respective domain. These particular hyperparameters have been chosen because of their potential to influence on the execution of logistic regression.

The "solver" hyperparameter is categorical, and its purpose is to specify the optimization method used to identify the optimal values for $\boldsymbol{\beta}$, which in turn should maximize the likelihood function. There are five different categories available for this hyperparameter: `"newton-cg"`,`"lbfgs"`, `"liblinear"`, `"sag"`, and `"saga"`. Further reading can be found in [14].

The "penalty" hyperparameter is also categorical and determines the method used to penalize the complexity of the model. There are three categories available for this hyperparameter: `"none"`,`"l1"`, and `"l2"`. These categories respectively correspond to no penalization on complexity, lasso regularization, and ridge regularization.

Table 5.1: The hyperparameters selected for tuning logistic regression, along with their default values, data type, and domain.

Hyperparameter	Default value	Data type	Domain
solver	`"lbfgs"`	categorical	`"newton-cg"`,`"lbfgs"`, `"liblinear"`, `"sag"`, `"saga"`
penalty	`"l2"`	categorical	`"none"`,`"l1"` , `"l2"`
tol	10^{-4}	float	$[0, \infty)$
max_iter	100	integer	$[1, \infty)$
C	1.0	float	$[0, \infty)$

The "tol" hyperparameter is continuous and sets the tolerance level for the stopping criteria of the algorithm. A lower value for "tol" will cause the algorithm to stop closer to the point of convergence. The "max_iter" hyperparameter is also continuous, and closely linked to the "tol" hyperparameter as it specifies the maximum number of iterations the algorithm can perform. Therefore, either "tol" or "max_iter" will determine the number of iterations executed by the algorithm.

The final hyperparameter chosen for logistic regression is "C". "C" is a continuous hyperparameter and represents the inverse of the regularization strength. A smaller value of "C" will thus result in a stronger regularization effect on the model's complexity.

5.1.2 Gradient Boosted Decision Trees

The modeling of gradient boosted decision trees involves using the `LGBMClassifier()` function imported from the `lightgbm` library. This predicting model has various hyperparameters that are listed in [9]. For this thesis, 10 hyperparameters were chosen for optimization, which are presented in table 5.2. All the hyperparameters selected for tuning in the gradient boosted decision trees are continuous variables.

The hyperparameter "learning_rate" serves as a step size to avoid overfitting. It reduces the feature weights after each boosting step, making the boosting process more conservative. Lower values of this hyperparameter will slow down the algorithm, but enhance its precision.

The hyperparameter "n_estimators" is closely linked to "learning_rate". It determines the number of boosting iterations, and therefore, the number of trees built. A small number of iterations may lead to an inaccurate model, while a large number may cause overfitting. When tuning this hyperparameter, it is necessary to take into account the value of "learning_rate".

Two hyperparameters that also are closely related to each other are "reg_alpha" and "reg_lambda". Both of these hyperparameters determine the extent to which the model's complexity is penalized. "reg_alpha" refers to L1 regularization, whereas "reg_lambda" refers to L2 regularization. Increasing the value of either of these hyperparameters results in a higher penalization of the models complexity.

The hyperparameter "min_child_samples" denotes the minimum sum of weights required to perform a split. If the resulting leaf node after the tree partitioning process has a weight sum that is lower than "min_child_samples", then the process of creating partitions will terminate. Reducing the value of "min_child_samples" will as a result lead to the construction of more complex trees.

"colsample_bytree" is a hyperparameter that functions similarly to bagging. It represents the subsample ratio of columns used to build each tree. For instance, if "colsample_bytree" is set to 0.6, 60% of the columns in the training data will be selected to construct each tree randomly. Subsampling occurs once for every tree created, and this process assists in extracting diverse insights from the data by restricting the number of features used in building each tree. Consequently, this approach may result in a more robust model.

Table 5.2: The hyperparameters selected for tuning gradient boosted decision trees, along with their default values, data type, and domain.

Hyperparameter	Default value	Data type	Domain
learning_rate	0.1	float	$(0, \infty)$
n_estimators	100	integer	$[0, \infty)$
reg_alpha	0	float	$[0, \infty)$
reg_lambda	0	float	$[0, \infty)$
min_child_samples	20	integer	$[0, \infty)$
colsample_bytree	1	float	$(0, 1]$
max_depth	-1	integer	$[0, \infty)$
subsample	1	float	$(0, 1]$
min_split_gain	0	float	$[0, \infty)$
num_leaves	31	integer	$(1, 131072]$

"max_depth" is an important hyperparameter that governs the maximum height of each tree in the model. A high value of this hyperparameter results in a complex model that is more susceptible to overfitting. The value of "max_depth" must be chosen carefully, taking into account the balance between model complexity and accuracy. One can notice that the default value of "max_depth" is -1, this means that there is no maximum depth set as a standard.

The hyperparameter "subsample" regulates bagging. It does this by specifying the proportion of training data used in constructing each tree. For instance, if "subsample" is set to 0.5, only half of the training data will be used in building each tree. Subsampling occurs randomly once for every tree created.

The hyperparameter "min_split_gain" decides the minimum gain required to execute a split. A higher value of this hyperparameter will lead to fewer splits in the tree, hence creating less complex models.

The hyperparameter "num_leaves" functions in a similar way to "max_depth", except that instead of specifying the maximum height of each tree, it sets a maximum number of leaf nodes. Similar to "max_depth", setting a high value for "num_leaves" will lead to a more complex model, which may increase the risk of overfitting.

5.1.3 Deep Learning

The sub-library `sklearn.neural_network` was used to model deep learning using the `MLPClassifier()` function. This predicting model has various hyperparameters that are described in [12]. For this thesis, 6 hyperparameters were chosen for optimization, which are presented in table 5.3.

The hyperparameter "activation" is categorical and decides the activation function used in the hidden layers in the neural network.

The hyperparameter "alpha" is continuous and decides the extent to which the model's complexity is penalized. This hyperparameter uses L2 regularization, and increasing the value will result in a higher penalization of the models complexity.

"learning_rate_init" is a continuous hyperparameter, and controls the initial learning rate used to train the model. It controls the step-size when updating the biases and weights in the neural network. A lower step-size will slow down the algorithm, but enhance its precision.

Table 5.3: The hyperparameters selected for tuning deep learning, along with their default values, data type, and domain.

Hyperparameter	Default value	Data type	Domain
activation	`"relu"`	categorical	`"relu"`,`"tanh"`, `"logistic"`, `"identity"`
alpha	0.0001	float	$[0, 1)$
learning_rate_init	0.001	float	$(0, 1)$
tol	0.0001	float	$(0, 1)$
max_iter	200	integer	$[1, \infty)$
hidden_layer_sizes	[100]	list of integers	$[1, \infty)$

The "tol" and "max_iter" hyperparameters serve the same purpose as for logistic regression. "tol" sets the tolerance level for the stopping criteria of the algorithm. A lower value for "tol" will cause the algorithm to halt closer to the point of convergence. The "max_iter" hyperparameter specifies the maximum number of iterations the algorithm can perform. Therefore, either "tol" or "max_iter" will determine the number of iterations executed by the algorithm.

Lastly, the hyperparameter called "hidden_layer_sizes" is responsible for determining the number of hidden layers and the number of neurons in each hidden layer in the neural network. It is specified as a list, so if one wish to create a model with three hidden layers and 20 neurons in each layer, one can set "hidden_layer_sizes" equal to "[20, 20, 20]". Increasing the number of layers and neurons per layer will result in a more complex model, which may improve prediction accuracy, but it also increases the risk of overfitting. For this predicting task, each hidden layer will contain the same amount of neurons in them. As a result, this hyperparameter creates two new hyperparameters, "num_layers" and "neurons_per_layer".

5.2 The procedure

The procedure for comparing the three predicting models involves six steps:

1. The dataset will be split into a training set and a test set. The predicting models will then be trained with default hyperparameters on the training set. Thereafter, the predicting models will be tested on the test set with a default threshold, and a comparison of the results will be made.

2. Cross-Validation and Bayesian optimization will be used on each of the predicting models as way to find optimal hyperparameters. This process will be well documented and a comparison between the the tuning process for the different predicting models will be discussed.

3. A second Bayesian optimization on the most important continuous hyperparameters of each of the predicting models will be performed. This is done to further investigate the relationships between some of the hyperparameters without the noise of the other hyperparameters, along with a re-justification of the search domains.

4. Cross-Validation will again be used, but now to find an optimal threshold for each of the predicting models.

5. The predicting models will again be trained on the training set, and then tested on the test set, but now with optimal hyperparameters and threshold. A comparison between the models will be made again.

6. Lastly, SHAP values will used as a way to compare feature importance across predicting models.

The splitting of the dataset into a training set and a test set is done using the function `train_test_split()` imported from the sub-library `sklearn.linear_selection`. The split is done randomly, and 75% of the data is used for training, and 25% is used for testing. This results in a training set of 197080 observations, and a test set of 65693 observations.

5.2.1 Bayesian Optimization

To perform Bayesian optimization, the first step involves creating an objective function that will be maximized, this is done using the syntax `def objective()`. Next, a "study" is created by calling the `optuna.create_study()` function. Finally, the study is used to optimize the objective value by using the `study.optimize()` function. The complete code can be found in Appendix G. It is crucial to choose an appropriate objective function to maximize. In this thesis, the same objective function will be used for all the predictive models to ensure that the tuning process can be compared across models.

In the objective, cross-validation is used with $N = 3$. This number is selected because it offers the advantages of cross-validation while remaining computationally efficient. Cross-validation with $N = 3$ will result in three groups of data, two with 65693 observations, and one with 65694 observations. As a consequence, three models are trained with identical hyperparameters but varying data. This is repeated for each iteration of the Bayesian optimization process.

In my project thesis [27], a similar optimization of hyperparameters was done. In that study, a threshold was also added to the optimization, this was done in case there was a link between the hyperparameters and the threshold, in the belief that it might yield better results than finding an optimal threshold later on. However, the dataset that is used now are approximately 100 times larger, and as a result, training one predictive model now is much more computational expensive. It leads to a significant higher training time, and hence a decrease in the iterations for the Bayesian optimization is necessary. The combination of fewer iterations and no significant improvement resulted in the decision to not include the threshold in the Bayesian optimization again.

As the threshold is left out of the Bayesian optimization, all the evaluation metrics have to be without a threshold. As a result, the objective value is chosen to be the mean of the AUC of the three predicting models. This metric is chosen as it is considered to be a good all-round test.

100 iterations with Bayesian optimization will be done. After interpreting the results, 50 more iterations will be conducted with fewer hyperparameters.

5.2.2 Threshold Optimization

After the optimal hyperparameters have been found, Cross-Validation with $N = 3$ is again used to find an optimal threshold for each predicting model. Each predicting model is now trained with optimal hyperparameters. An objective is then formed as a way to tell how good the threshold being tested is. A list of 1000 evenly spaced points between 0 and 1 is then used to test the objective value for every threshold. The objective value is defined as

$$\text{Objective value} = \sum_{i=1}^{3} \big(\text{MCC}_i + 2 \cdot \text{BACC}_i\big), \tag{5.1}$$

where MCC is the Matthews Correlation Coefficient and BACC is Balanced Accuracy. The reason BACC is multiplyed with 2 is that it ranges from 0 to 1, whereas MCC ranges from -1 to 1, so multiplying with 2 gives more balance between the metrics. These metrics are used as they both are good all-round metrics on imbalanced datasets.

5.2.3 SHAP values

The SHAP values for each predicting model, both with and without tuned hyperparameters, will be found so that there can be done an comparison of the feature importance across predicting models. Both the `shap.KernelExplainer()` function and the `explainer.shap_values()` function, from the `shap` library [30], will be used to do this.

Chapter 6

Analysis and Results

The analysis and results obtained from the techniques described in chapter 5 will be presented in this chapter. First, the results obtained using predictive models with default hyperparameters and threshold will be presented. Subsequently, the findings from Bayesian optimization, including the optimal threshold for each predictive model, will be shown. Next, the results achieved with predictive models using optimal hyperparameters and threshold will be presented. In the end, an analysis of the SHAP values for each predictive model will be provided.

The whole dataset is used for the prediction task. In the dataset there are multiple observations per customer, however the feature "BK_ACCOUNT_ID", which is the personal account number of a customer, is removed. As a result, each observation is assumed to be independent. The prediction task is made so that there will be done predictions for a single customer at several different months. This approach results in an incorrect prediction if a customer is classified as becoming active in a month when they actually remain passive, even though they become active in a later month.

6.1 Results with default hyperparameters

The predictive models were first trained on the training set using default hyperparameters. Thereafter, the models were evaluated on the test set, with a threshold value of 0.064. This particular threshold was chosen because it corresponds to the proportion of active and passive customers in the dataset. Previous studies [3] suggest that using this threshold as a default value may be more effective than simply relying on 0.5.

Table 6.1, 6.2, and 6.3 display the confusion matrices showing how each predictive model categorized the test set. The predictions of the three models are comparable. Notably, deep learning assigns the "becoming active" label to customers less often than the other models. On the other hand gradient boosted decision trees seems to assign the "becoming active" label to customers slightly more often than logistic regression.

Table 6.1: Confusion matrix from logistic regression. The training of the model was performed on the training set using default hyperparameters, and the model was evaluated on the test set using a default threshold of 0.064. 0 represents remaining passive while 1 represents becoming active.

True \ Predicted	0	1
0	36028	25468
1	2010	2188

Table 6.2: Confusion matrix from gradient boosted decision trees. The training of the model was performed on the training set using default hyperparameters, and the model was evaluated on the test set using a default threshold of 0.064. 0 represents remaining passive while 1 represents becoming active.

True \ Predicted	0	1
0	32714	28782
1	1362	2836

Table 6.3: Confusion matrix from deep learning. The training of the model was performed on the training set using default hyperparameters, and the model was evaluated on the test set using a default threshold of 0.064. 0 represents remaining passive while 1 represents becoming active.

True \ Predicted	0	1
0	45174	16322
1	2741	1457

Table 6.4 displays the results of each classification metric. It is important to note that the Brier score is the only metric where minimization is the goal. Consequently, to transform the objective of all metrics presented in the table into maximization, 1−Brier score is employed.

One can see that gradient boosted decision trees outperforms both logistic regression and deep learning in every all-round metric. The only metrics where the other predictive models shows better results are in accuracy and sensitivity. Nonetheless, accuracy and sensitivity alone does not provide a complete picture of the overall prediction performance.

Logistic regression and deep learning performs quite similar in all the all-round metrics. However, logistic regression does a slightly better predicting job overall.

Table 6.4: Results from classification metrics on the test set for all models with default hyperparameters and threshold.

Predicting model	AUC	1−Brier score	MCC	BACC	Accuracy	Sensitivity	Specificity
Logistic regression	0.5731	0.9403	0.0530	0.5535	0.5817	0.5859	0.5212
Gradient boosted decision trees	0.6461	0.9407	0.1016	0.6038	0.5411	0.5320	0.6756
Deep learning	0.5580	0.9302	0.0450	0.5408	0.7098	0.7346	0.3471

6.1.1 Investigating threshold importance

The used threshold is not optimized, and it may be advantageous to examine how different evaluation metrics are affected by it. Figure 6.1, 6.2, and 6.3 visualizes this. These figures indicates that the MCC is slightly less sensitive to the threshold in comparison to the BACC. Moreover, it becomes evident that the accuracy metric is not an ideal test for unbalanced datasets.

The ideal threshold for each predictive model remains relatively consistent across all metrics except for the accuracy. The optimal threshold for the MCC and the BACC is roughly the same as the proportion of active and passive customers in the dataset. One can therefore conclude that choosing 0.064 as a default threshold is not that bad. For accuracy, the optimal threshold seems to be the higher the better.

It is noteworthy that even though the maximum MCC and BACC scores of deep learning are not as high as those of the other predictive models, the scores of deep learning appear to be less affected by changes in the threshold value.

Appendix F contains similar graphs, although these focus on Sensitivity and Specificity instead.

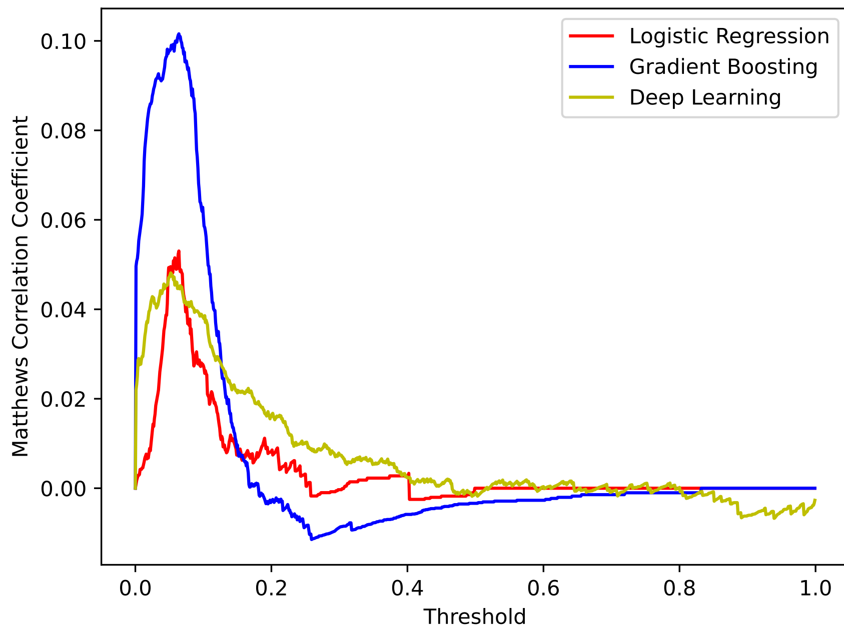

Figure 6.1: A plot illustrating the impact of various threshold values on the Matthews Correlation Coefficient (MCC) for different predictive models with default hyperparameters. Logistic regression is displayed in red, gradient boosted decision trees is displayed in blue and deep learning is displayed in yellow.

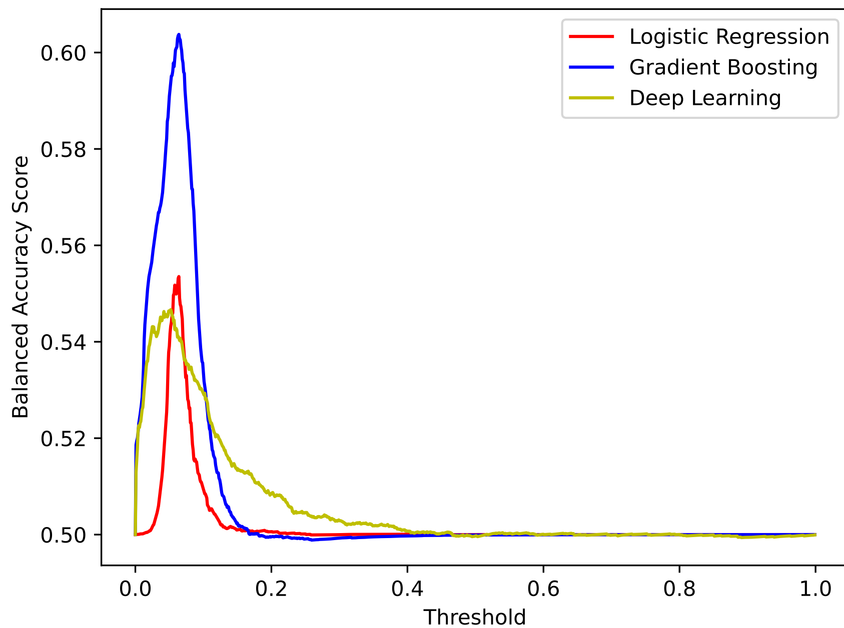

Figure 6.2: A plot illustrating the impact of various threshold values on the Balanced Accuracy (BACC) for different predictive models with default hyperparameters. Logistic regression is displayed in red, gradient boosted decision trees is displayed in blue and deep learning is displayed in yellow.

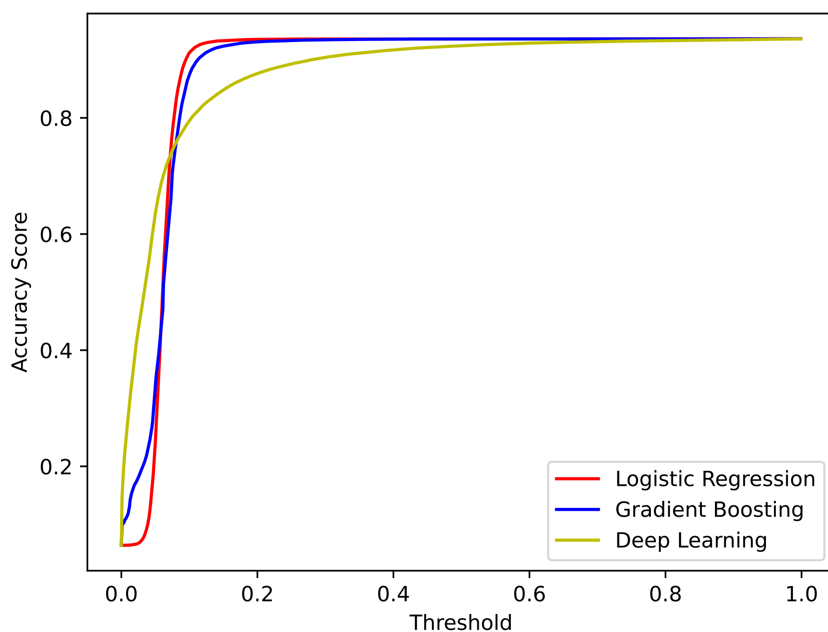

Figure 6.3: A plot illustrating the impact of various threshold values on the Accuracy for different predictive models with default hyperparameters. Logistic regression is displayed in red, gradient boosted decision trees is displayed in blue and deep learning is displayed in yellow.

6.1.2 Feature Importance

Figure 6.4 and 6.5 exhibit the feature importance for each predictive model. The feature importance for logistic regression is determined by the absolute value of the coefficients β_j. The scale is set so that the feature with the highest coefficient has a feature importance of 100.

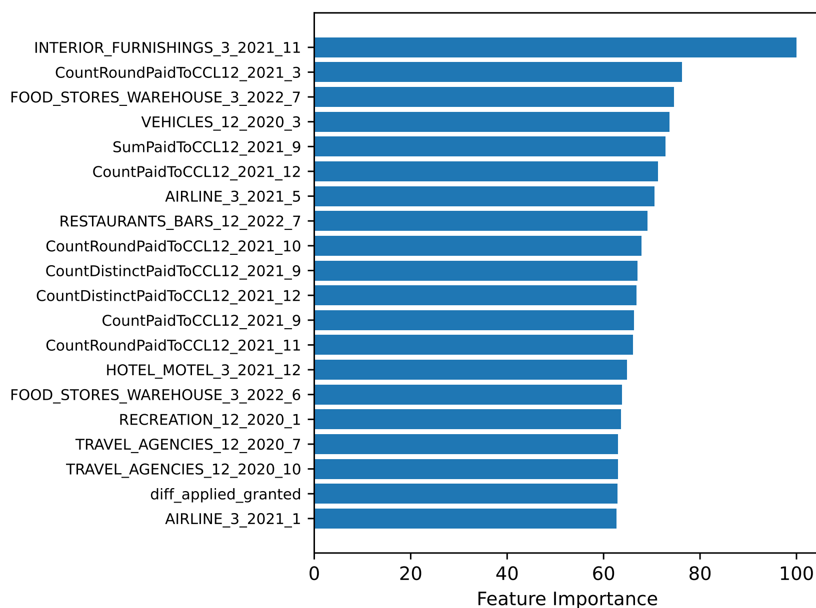

Figure 6.4: Feature importance for logistic regression with default hyperparameters. The top 20 most important features are displayed.

The feature importance regarding gradient boosted decision trees is evaluated based on the improvement of the performance measure attributed to each split in a tree. This is weighted by the number of observations for which each node is responsible for. The overall feature importance is determined by averaging the feature importance across all the trees in the model.

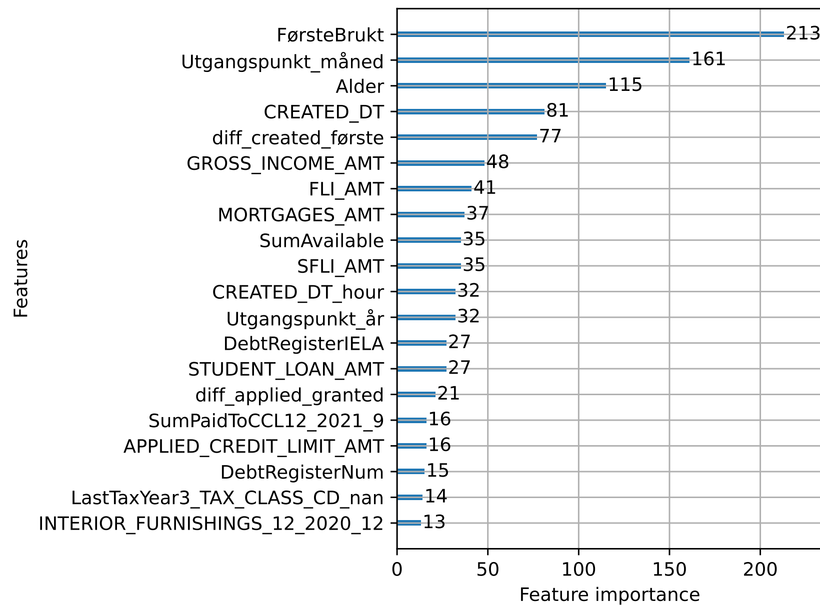

Figure 6.5: Feature importance for gradient boosted decision trees with default hyperparameters and threshold. The top 20 most important features are displayed.

One can see that logistic regression and gradient boosted decision trees do not share any of the top 20 most important features except "SumPaidToCCL12_2021_9" and "diff_applied_granted". "SumPaidToCCL12_2021_9" is the number 5 most important feature for logistic regression, whereas it is the number 15 in gradient boosted decision trees. One can also see that the top 20 most important features for logistic regression only contain historical credit card uses or transactions except "diff_applied_granted". "diff_applied_granted" is not in the top 10 most important features for any of the two predicting models.'

Other than that "INTERIOR_FURNISHINGS_3_2021_11", "CountRoundPaidToCCL12_2021_3" and "FOOD_STORES_WAREHOUSE_3_2022_7" are the most important features for logistic regression. Whereas "FørsteBrukt", "Utgangspunkt_måned" and "Alder" are the most important features in gradient boosted decision trees.

The feature importance for deep learning is not provided in the Scikit-learn package, there will as a result be looked at SHAP values for deep learning. This will be done in the last section of this chapter. The SHAP values for the other predicting models will also be included.

Outprint from the code is included in Appendix F.

6.2 Logistic Regression Optimization

In this section of the analysis and results, there will only be used the training dataset. Cross-validation is employed to evaluate the different combinations of hyperparameters and threshold values. This is done to simulate real world scenarios, where the building and training of the predictive models has to be done without any knowledge of the test dataset.

Logistic regression does not allow for the execution of every possible hyperparameter combination. This is because certain variables in the "solver" and "penalty" options may not be compatible. For

instance, the ”solver” `"lbfgs"` is not able to use the ”penalty” `"l1"`. As a result, the ”penalty” `"l1"` along with the ”solver” `"liblinear"` will not be used in the optimization. Table 6.5 displays the hyperparameters chosen for optimization along with their search domains.

Table 6.5: The hyperparameters with their search domain for the first logistic regression hyperparameter tuning.

Hyperparameter	Search Domain
solver	`"newton-cg"`,`"lbfgs"`, `"sag"`, `"saga"`
penalty	`"none"` , `"l2"`
tol	$[10^{-10}, 10^{-2}]$
max_iter	$[10, 100]$
C	$[0.05, 100]$

6.2.1 First Results

Table 6.6 shows the outcome of the first hyperparameter tuning process. The tuning took 10 hours and 36 minutes to execute and yielded an AUC value of 0.5697.

Table 6.6: The hyperparameters with their optimal values for the first logistic regression hyperparameter tuning.

Hyperparameter	Value
solver	`"saga"`
penalty	`"l2"`
tol	$3.8727 \cdot 10^{-8}$
max_iter	35
C	0.2505

Figure 6.6 shows a history plot of the first Bayesian optimization. This plot displays the objective value (AUC) for each iteration of the Bayesian optimization process. It is evident that the best objective value shows some improvement in the beginning, but eventually plateaus after around 20 iterations.

Figure 6.7 shows the degree of importance that each hyperparameter has on the objective values seen in Figure 6.6. The plot indicates that ”solver” is the most significant hyperparameter, followed by ”penalty” and ”C”. In contrast, the other two hyperparameters, ”max_iter” and ”tol”, appear to have minimal impact on the objective value. When examining Figure 6.8, it becomes evident why ”solver” is of such high importance: selecting an inappropriate solver can significantly compromise the predictive performance of the model.

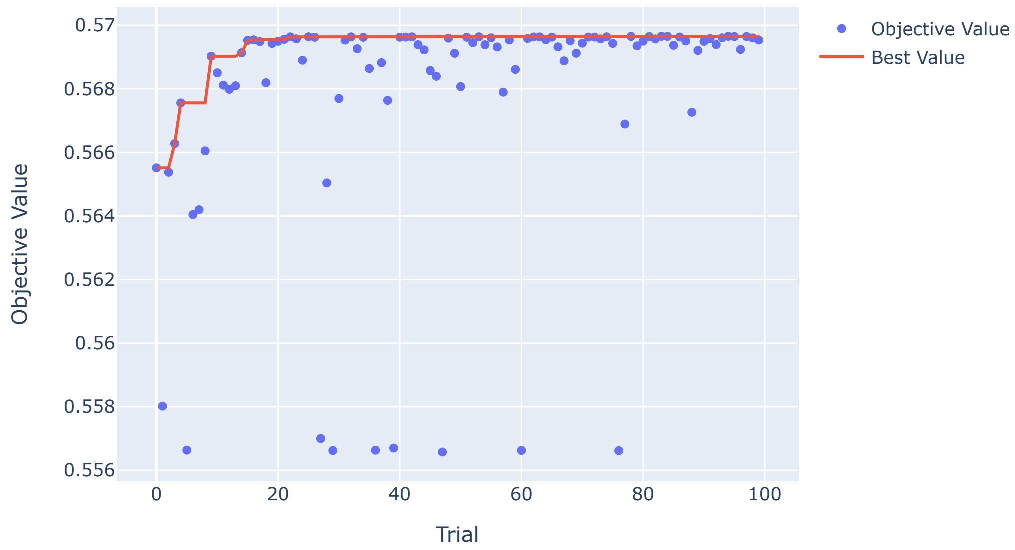

Figure 6.6: Optimization history plot of the Bayesian optimization. The objective value (AUC) for each iteration are denoted by blue dots. The red line corresponds to the best objective value obtained so far.

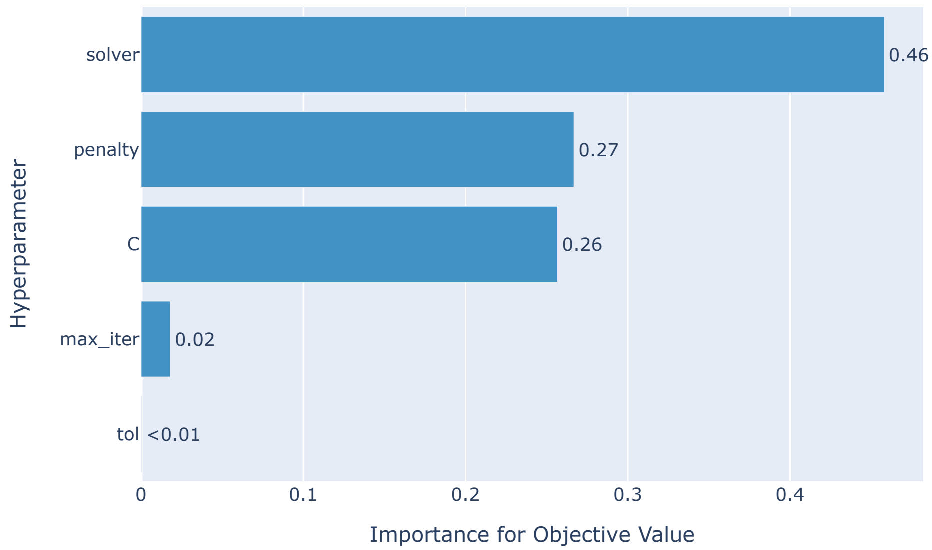

Figure 6.7: Hyperparameter importance plot indicating the importance each hyperparameter has on the objective value.

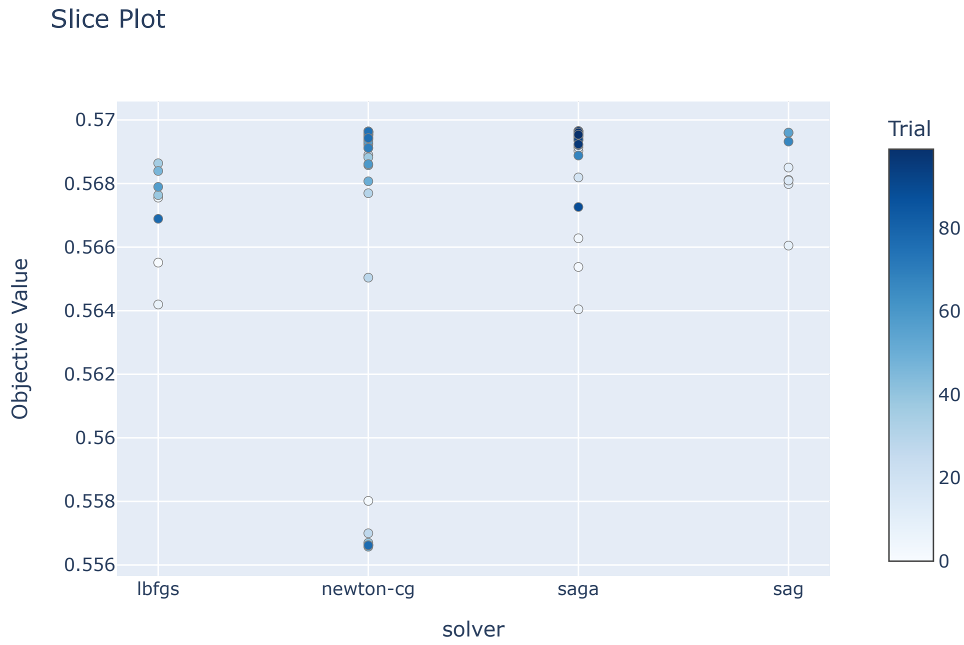

Figure 6.8: Slice plot showing the impact the hyperparameter "solver" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

Figure 6.9 indicates that the upper bound on the objective value is influenced by the "C" hyperparameter. This suggests that determining the ideal degree of penalizing complexity is crucial for achieving optimal results.

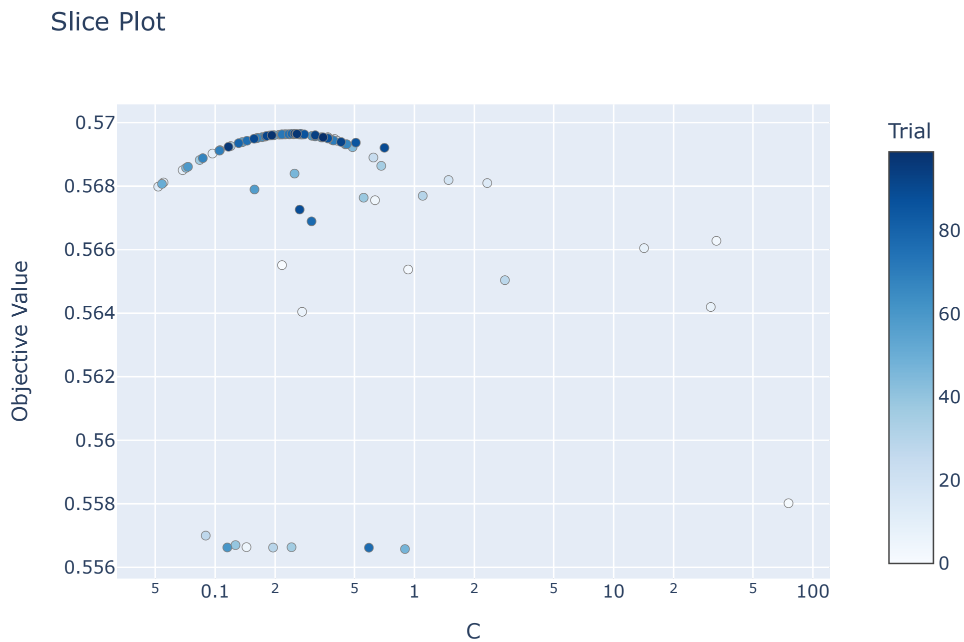

Figure 6.9: Slice plot showing the impact the hyperparameter "C" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

6.2.2 Second Results

To further investigate the importance of the hyperparameters, a second Bayesian optimization has been done with 50 iterations. The second Bayesian optimization will only focus on the continuous hyperparameters and not the categorical. Additionally, the domains of these hyperparameters have been adjusted. Table 6.7 displays the hyperparameters chosen for a second optimization along with their new search domains.

Table 6.7: The hyperparameters with their respective search domain for the second logistic regression hyperparameter tuning.

Hyperparameter	Search Domain
max_iter	$[10, 80]$
tol	$[10^{-6}, 10^{-3}]$
C	$[0.01, 10]$

Table 6.8 shows the outcomes of the hyperparameter tuning process. The second hyperparameter tuning took 2 hours and 59 minutes to execute and yielded an AUC value of 0.5697. This means that there was not any improvement in the objective value for the second optimization compared to the first one.

Table 6.8: The hyperparameters with their optimal values for the second logistic regression hyperparameter tuning.

Hyperparameter	Value
max_iter	54
tol	$1.3111 \cdot 10^{-4}$
C	0.2333

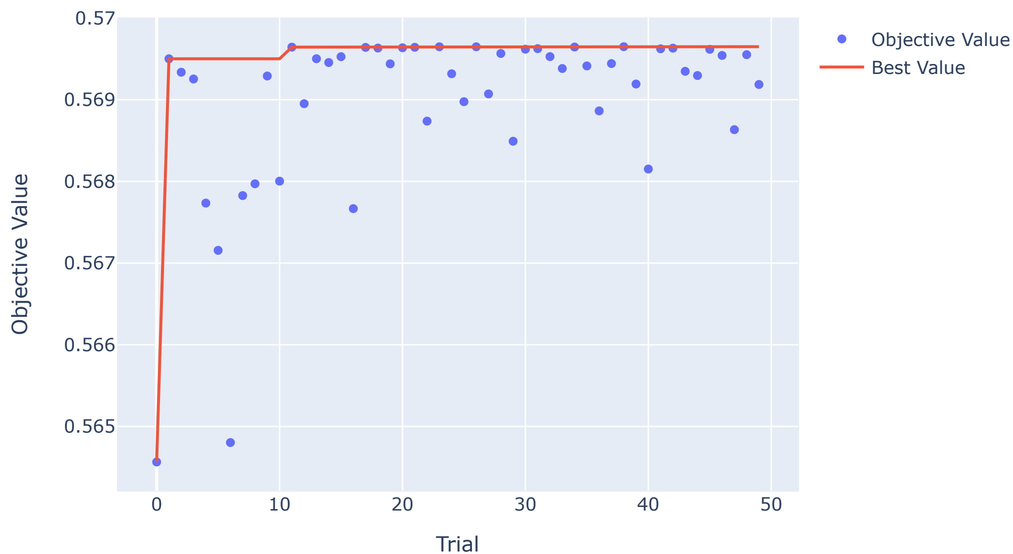

Figure 6.10: Optimization history plot of the Bayesian optimization. The objective values (AUC) for each iteration are denoted by blue dots. The red line corresponds to the best objective value obtained so far.

Figure 6.10 shows a history plot of the second Bayesian optimization. It is evident that the best objective value shows some improvement in the beginning, but eventually plateaus after 10 to 15 iterations.

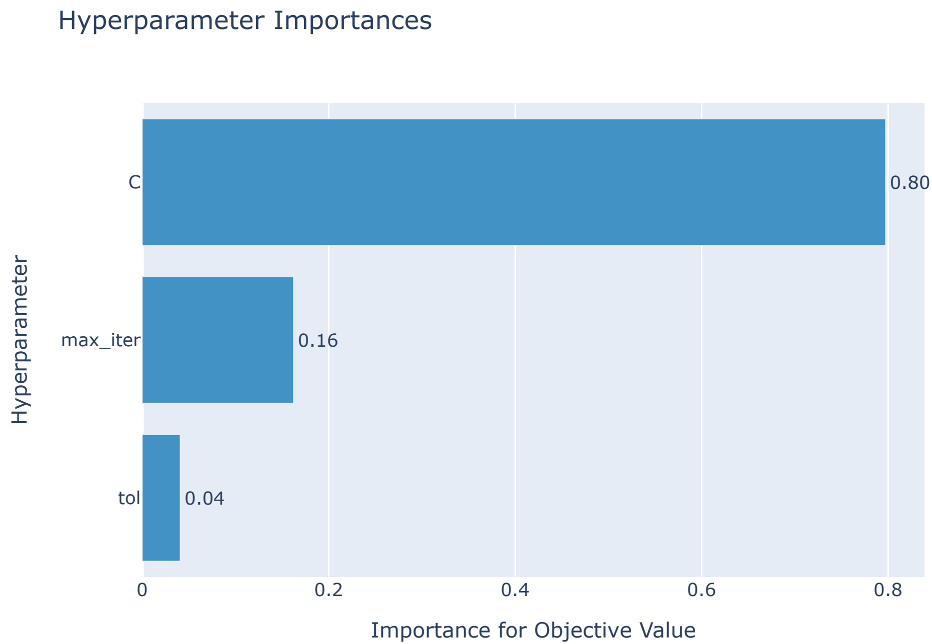

Figure 6.11: Hyperparameter importance plot indicating the importance each hyperparameter has on the objective value.

Figure 6.11 shows the degree of importance that each hyperparameter has on the objective value. The plot indicates that "C" is the most significant hyperparameter. "max_iter" appear to have some impact, and "tol" appear to have a minimal impact on the objective value.

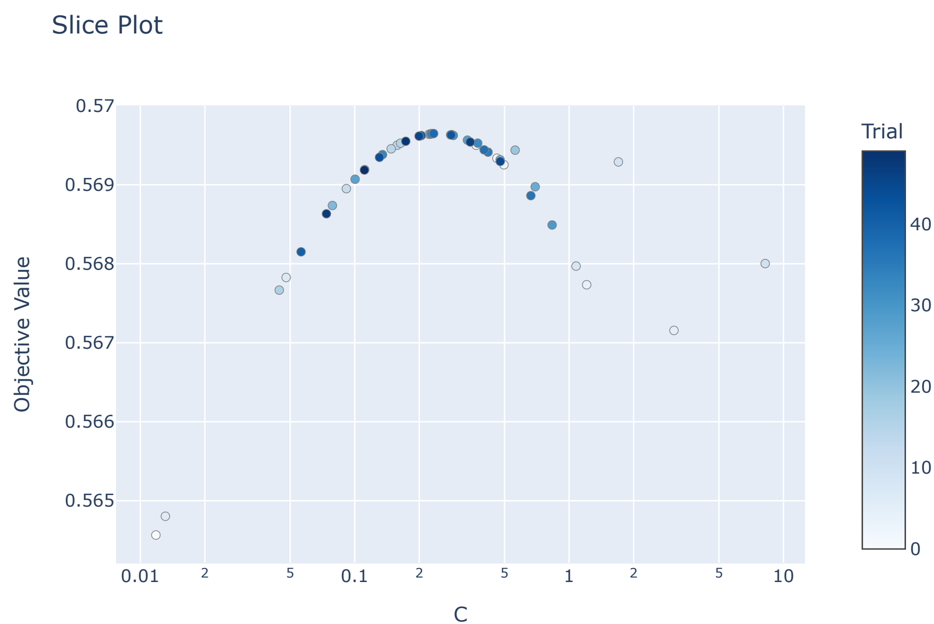

Figure 6.12: Slice plot showing the impact the hyperparameter "C" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

When examining Figure 6.12, it becomes evident why "C" is so important. In that figure, it can seem as if the objective value is almost a function of "C" only.

Figure 6.13 shows a Contour plot. In this plot one can see how two different hyperparameters influences the objective value. Darker color represents better objective value. It is clear that "C" has the highest influence on the objective value.

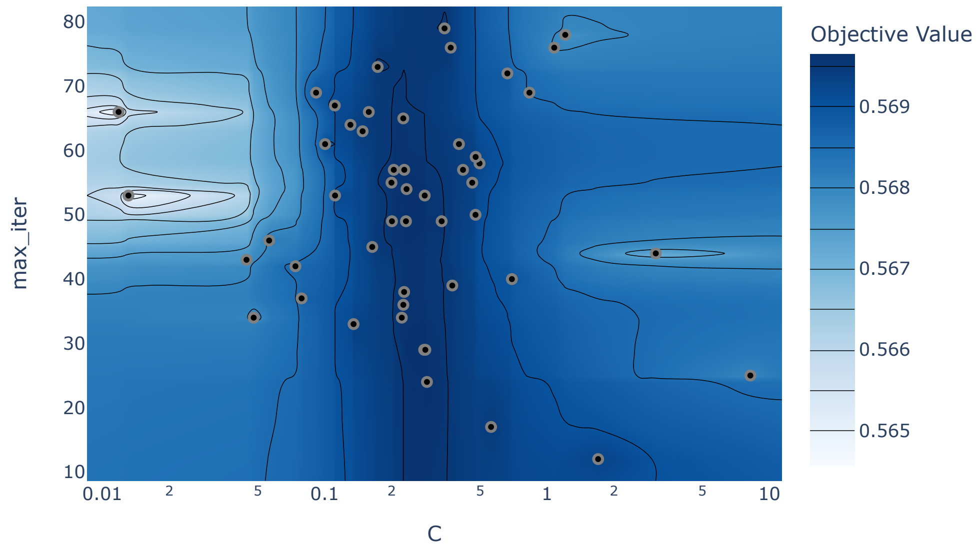

Figure 6.13: Contour plot showing how different combinations of "C" and "max_iter" changes the objective value. Darker color represents better objective values.

Outprint from the code as well as the slice plots for the other hyperparameters are included in Appendix C.

6.2.3 Threshold

The logistic regression model with optimal hyperparameters achieved its best performance at a threshold of 0.056. Figure 6.14 illustrates the impact of varying the threshold between zero and one on the objective value defined in Chapter 5. It is evident from the graph that there is a distinct optimal threshold that maximizes the model's performance.

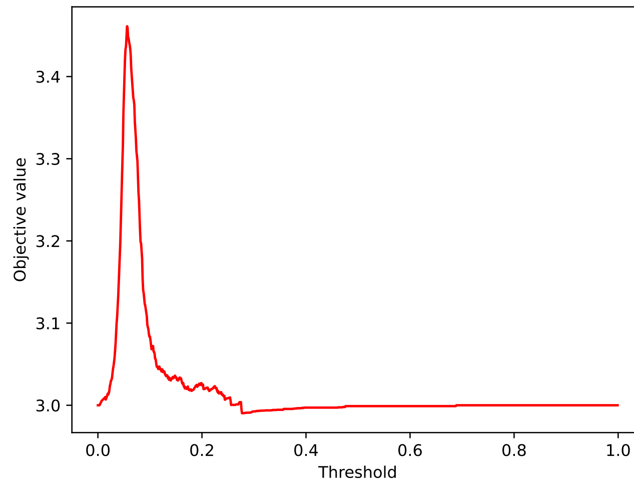

Figure 6.14: A plot showing how different values of the threshold affect an objective value.

6.3 Gradient Boosted Decision Trees Optimization

In this section of the analysis and results, there will only be used the training dataset. Cross-validation is employed to evaluate the different combinations of hyperparameters and threshold values. This is done to simulate real world scenarios, where the building and training of the predictive models has to be done without any knowledge of the test dataset.

Table 6.9 displays the domains that have been chosen for the first hyperparameter search along with the respective hyperparameter.

Table 6.9: The hyperparameters with their respective search domain for the first gradient boosted decision trees hyperparameter tuning.

Hyperparameter	Search Domain
learning_rate	$[0.001, 0.3]$
n_estimators	$[10, 3000]$
reg_alpha	$(0, 50]$
reg_lambda	$(0, 50]$
min_child_samples	$[0, 300]$
colsample_bytree	$[0.01, 1]$
max_depth	$[3, 20]$
subsample	$[0.01, 1]$
min_split_gain	$[0.01, 1]$
num_leaves	$[10, 5000]$

6.3.1 First Results

Table 6.10 shows the outcome of the first hyperparameter tuning process. The tuning took 8 hours and 31 minutes to execute and yielded an AUC value of 0.6569.

One thing to notice is that the value of the ”learning_rate” became quite close to the edge of the search domain. In Figure 6.17 one can see that the points are focused towards a lower ”learning_rate”. As a result, the second Bayesian optimization will include this hyperparameter with a domain containing smaller values.

Table 6.10: The hyperparameters with their optimal values for the first gradient boosted decision trees hyperparameter tuning.

Hyperparameter	Optimal value
learning_rate	0.001018
n_estimators	2272
reg_alpha	0.003702
reg_lambda	0.7099
min_child_samples	57
colsample_bytree	0.2303
max_depth	20
subsample	0.5025
min_split_gain	0.9187
num_leaves	759
threshold	0.3313

Figure 6.15 shows a history plot of the first Bayesian optimization. This plot displays the objective value (AUC) for each iteration of the Bayesian optimization process. It is evident that the best objective value shows substantial improvement in the beginning, but eventually plateaus after around 30 iterations.

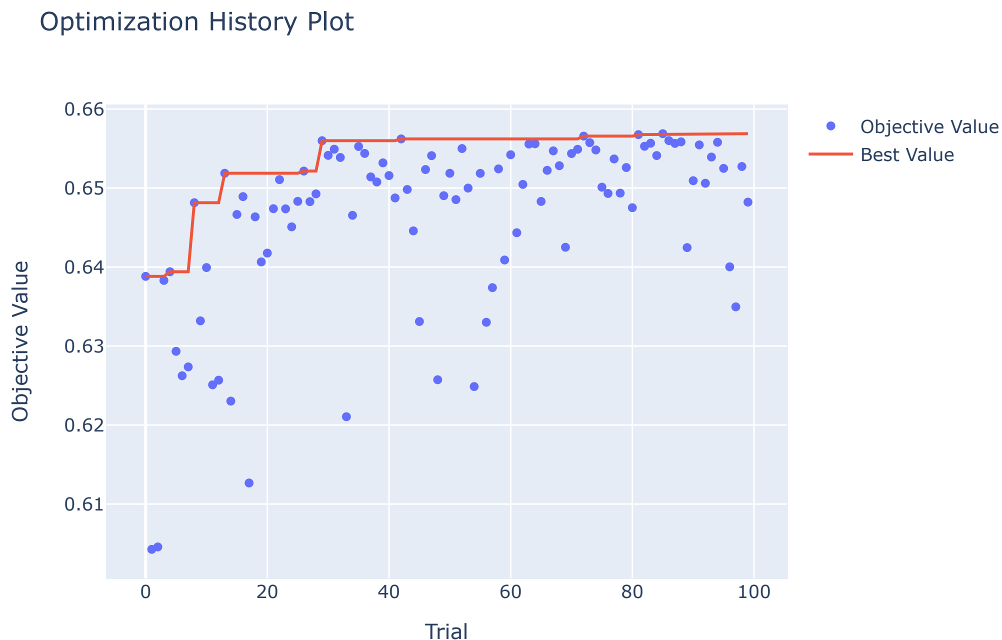

Figure 6.15: Optimization history plot of the Bayesian optimization. The objective values (AUC) for each iteration are denoted by blue dots. The red line corresponds to the best objective value obtained so far.

Figure 6.16 shows the degree of importance that each hyperparameter has on the objective value. The plot indicates that ”learning_rate” is the most significant hyperparameter, followed closely by ”min_split_gain”. The other hyperparameters appear to have some to minimal impact on the objective value. When examining Figure 6.17, it becomes evident why ”learning_rate” is of such high importance; a lower value for ”learning_rate” clearly gives better Objective values.

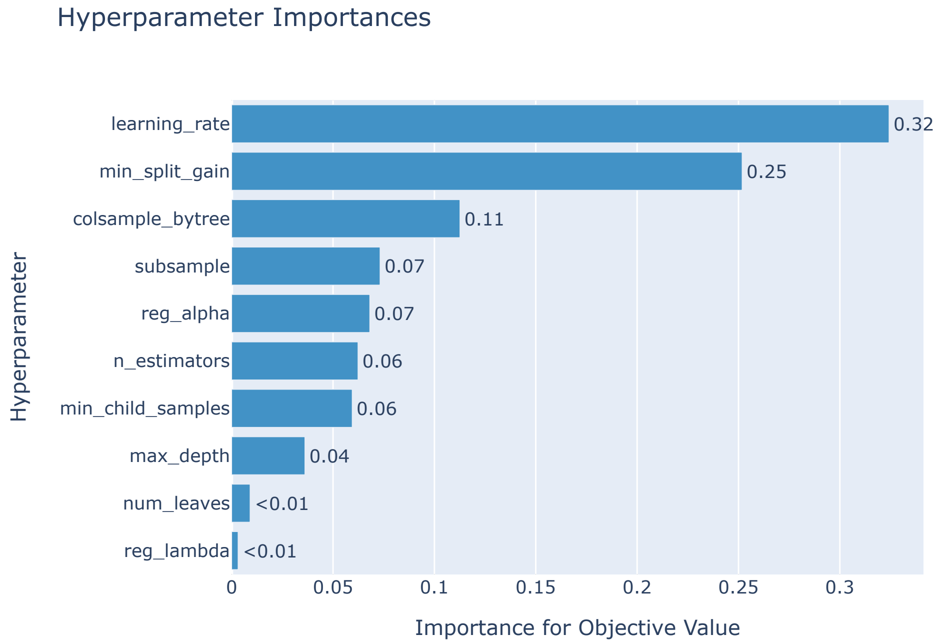

Figure 6.16: Hyperparameter importance plot indicating the importance each hyperparameter has on the objective value.

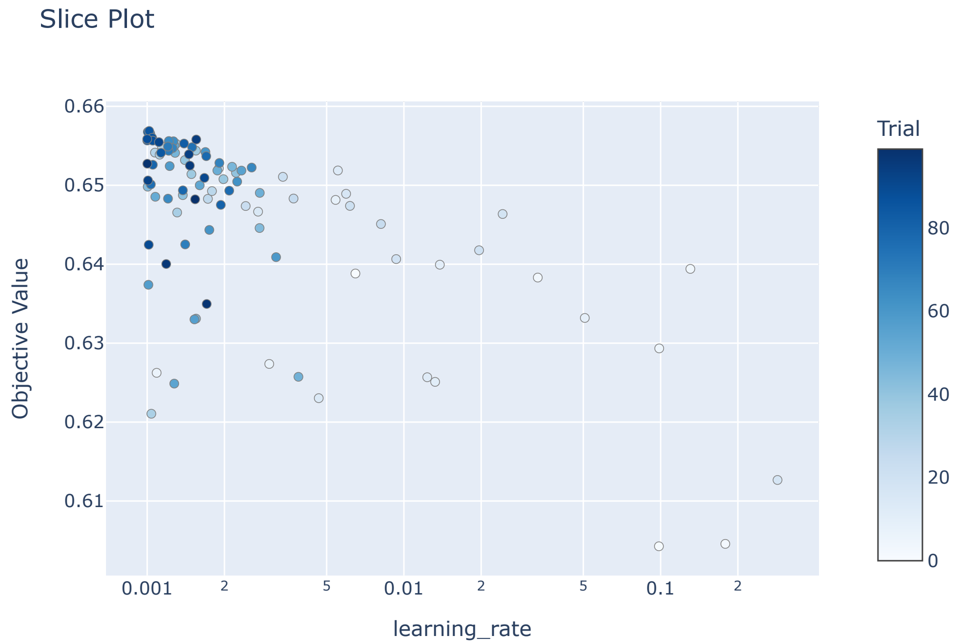

Figure 6.17: Slice plot showing the impact the hyperparameter ”learning_rate” has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

Figure 6.18 provides the importance of ”min_split_gain”. The plot displays how varying values of ”min_split_gain” changes the corresponding objective values. The results demonstrate that there exists an optimal range between 0.8 and 1.

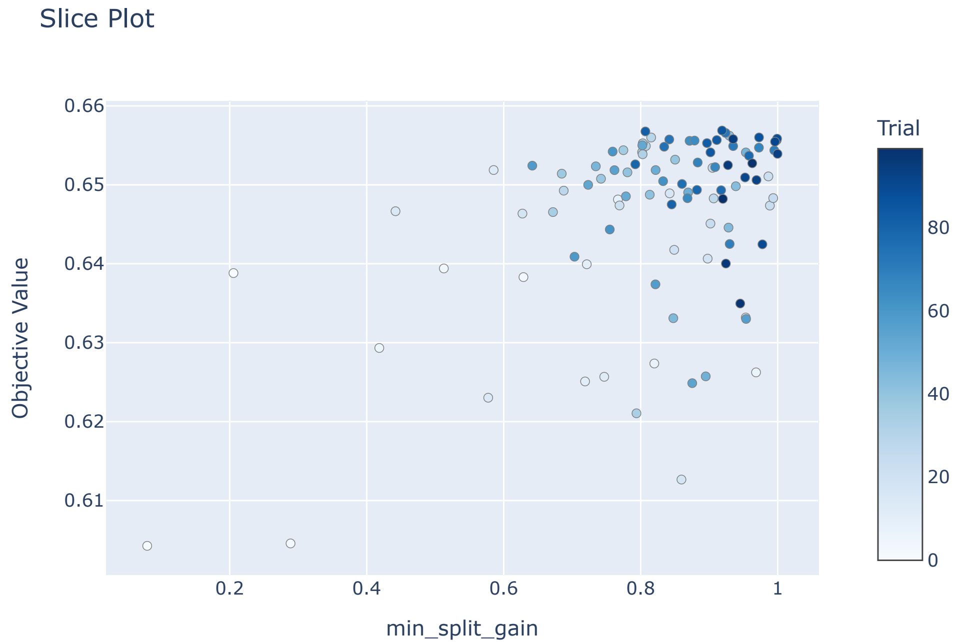

Figure 6.18: Slice plot showing the impact the hyperparameter ”min_split_gain” has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

6.3.2 Second Results

To further investigate the importance of the hyperparameters, a second Bayesian optimization has been done with 50 iterations.

There are a lot of hyperparameters to choose from in gradient boosted decision trees, but three hyperparameters have been selected for the second optimization. Table 6.11 displays these hyperparameters with their respective domain. ”learning_rate” and ”max_depth” are chosen because the search domains of these hyperparameters needs some readjustments. ”n_estimators” is chosen as there could be a tendency that if the step size hyperparameter ”learning_rate” gets smaller, there might be needed to build more trees to get an optimal combination of hyperparameters.

Table 6.11: The hyperparameters with their respective search domain for the second gradient boosted decision trees hyperparameter tuning.

Hyperparameter	Search Domain
learning_rate	$[0.0001, 0.003]$
n_estimators	$[2000, 5000]$
max_depth	$[15, 30]$

Table 6.12 shows the outcomes of the hyperparameter tuning process. The second hyperparameter tuning took 11 hours and 28 minutes to execute and yielded an AUC value of 0.6592. This means that there was an improvement of 0.0023 in the objective value for the second optimization compared to the first one.

Table 6.12: The hyperparameters with their optimal values for the second gradient boosted decision trees hyperparameter tuning.

Hyperparameter	Value
learning_rate	0.0005807
n_estimators	2789
max_depth	30

Figure 6.19 shows a history plot of the second Bayesian optimization. It is evident that the best objective value shows larger improvements in the beginning, but eventually slows down, but still shows some improvement even after 40 iterations.

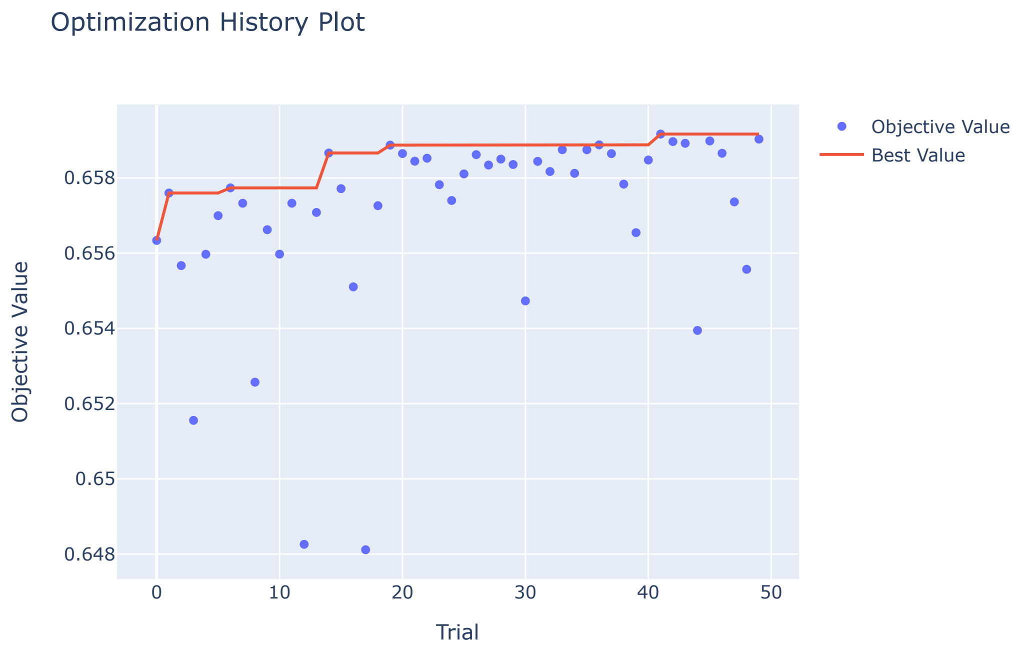

Figure 6.19: Optimization history plot of the Bayesian optimization. The objective values (AUC) for each iteration are denoted by blue dots. The red line corresponds to the best objective value obtained so far.

Figure 6.20 shows the degree of importance that each hyperparameter has on the objective value. The plot indicates that "learning_rate" is the most significant hyperparameter. "max_depth" and "n_estimators" appear to have some impact on the objective value.

Figure 6.21 provides a visual representation of the importance of "learning_rate". The plot displays varying values of "learning_rate" and their corresponding objective values. The results demonstrate that there exists an optimal range between 0.0007 and 0.0003.

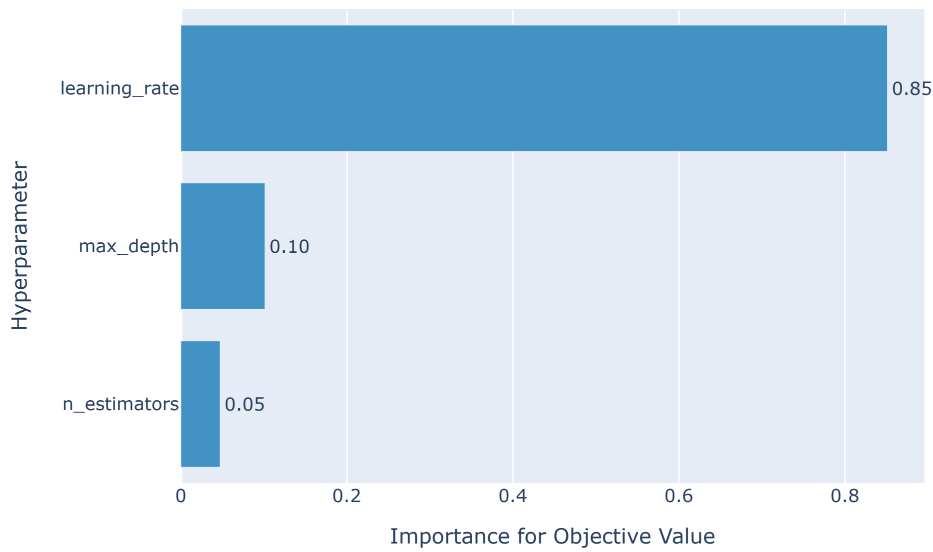

Figure 6.20: Hyperparameter importance plot indicating the importance each hyperparameter has on the objective value.

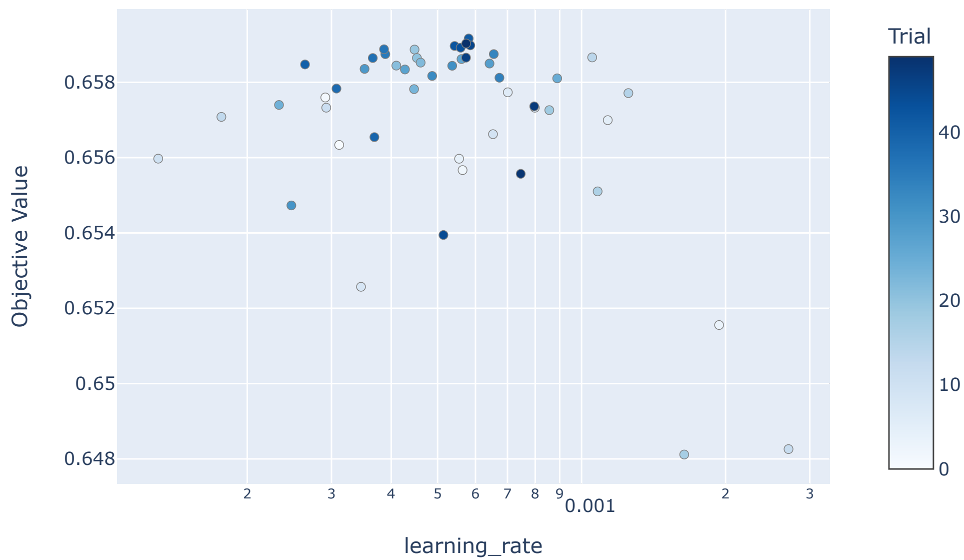

Figure 6.21: Slice plot showing the impact the hyperparameter "learning_rate" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

Figure 6.22 shows a Contour plot. In this plot one can see how two different hyperparameters influences the objective value. Darker color represents better objective value. It is clear that both "learning_rate" and "max_depth" plays a significant role on the model performance.

Outprint from the code as well as the slice plots for the other hyperparameters are included in Appendix D.

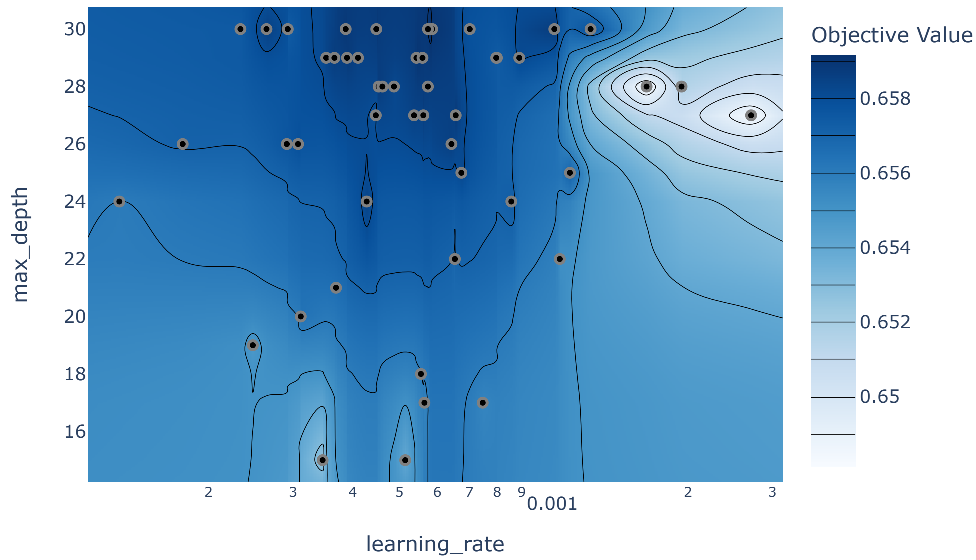

Figure 6.22: Contour plot showing how different combinations of "learning_rate" and "max_depth" changes the objective value. Darker color represents better objective values.

6.3.3 Threshold

The gradient boosted decision trees model with optimal hyperparameters achieved its best performance at a threshold of 0.060. Figure 6.23 illustrates the impact of varying the threshold between zero and one on the objective value defined in Chapter 5. It is evident from the graph that there is a distinct optimal threshold that maximizes the model's performance.

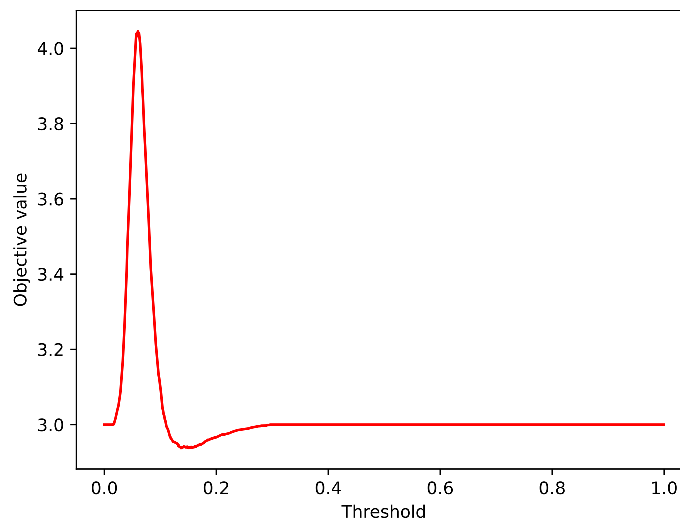

Figure 6.23: A plot showing how different values of the threshold affect an objective value.

6.4 Deep Learning Optimization

In this section of the analysis and results, there will only be used the training dataset. Cross-validation is employed to evaluate the different combinations of hyperparameters and threshold values. This is done to simulate real world scenarios, where the building and training of the predictive models has to be done without any knowledge of the test dataset.

Table 6.13 displays the hyperparameters chosen for the first Bayesian optimization along with their respective search domain.

Table 6.13: The hyperparameters with their respective search domain for the first deep learning hyperparameter tuning.

Hyperparameter	Search Domain
activation	`"relu"`,`"tanh"`, `"logistic"`, `"identity"`
alpha	$[10^{-8}, 1]$
learning_rate_init	$[10^{-5}, 1]$
tol	$[10^{-6}, 0.1]$
max_iter	$[2, 100]$
num_layers	$[2, 20]$
neurons_per_layer	$[2, 200]$

6.4.1 First Results

Table 6.14 shows the outcome of the first hyperparameter tuning process. The tuning took 53 hours and 6 minutes to execute and yielded an AUC value of 0.5917.

One thing to notice is that the optimal value of ”num_layers” became 2, which is on the edge of the search domain. In Figure 6.26 one can see that lower values for ”num_layers” often results in a higher objective value. This will be further tested in the second Bayesian optimization.

Table 6.14: The hyperparameters with their optimal values for the first deep learning hyperparameter tuning.

Hyperparameter	Optimal value
activation	`"logistic"`
alpha	$9.1556 \cdot 10^{-4}$
learning_rate_init	$2.3164 \cdot 10^{-3}$
tol	$1.3651 \cdot 10^{-6}$
max_iter	62
num_layers	2
neurons_per_layer	169

Figure 6.24 shows a history plot of the first Bayesian optimization. This plot displays the objective value (AUC) for each iteration of the Bayesian optimization process. It is evident that the best objective value shows substantial improvement even after 60 iterations, but eventually plateaus. It is also noteworthy that deep learning produces many models that performs as bad as a random classifier, and some that perform even worse. It is thus important to choose good hyperparameters when using deep learning.

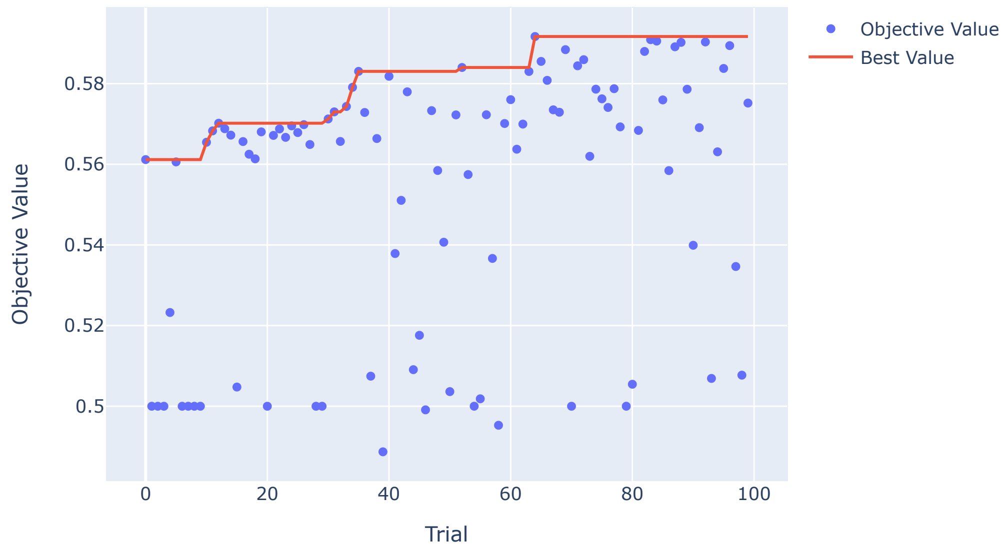

Figure 6.24: Optimization history plot of the Bayesian optimization. The objective values (AUC) for each iteration are denoted by blue dots. The red line corresponds to the best objective value obtained so far.

Figure 6.25 shows the degree of importance that each hyperparameter has on the objective value. The plot indicates that "num_layers" is the most significant hyperparameter, followed by "activation" and "max_iter". The other hyperparameters appear to have some impact to minimal impact on the objective value. When examining Figure 6.26, it becomes evident why "num_layers" is of such high importance; a lower value for "num_layers" can give better objective values.

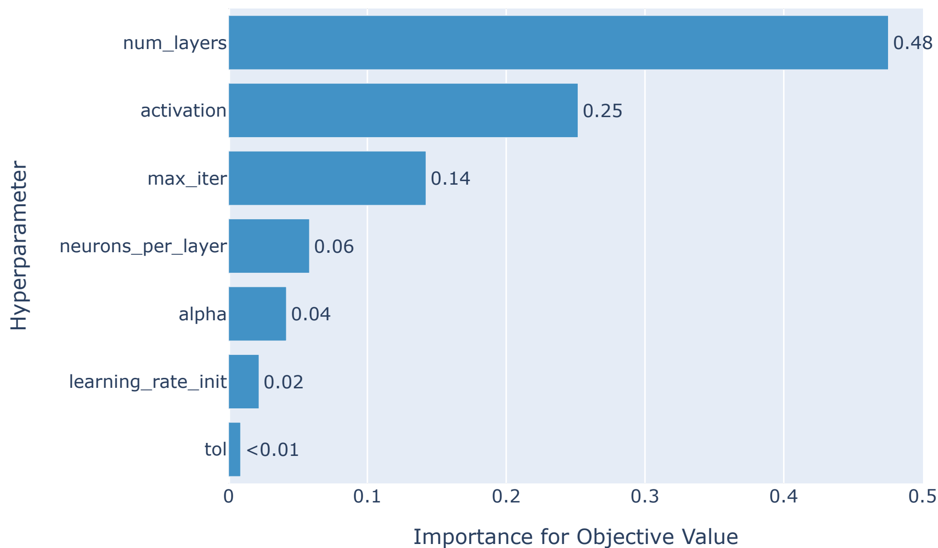

Figure 6.25: Hyperparameter importance plot indicating the importance each hyperparameter has on the objective value.

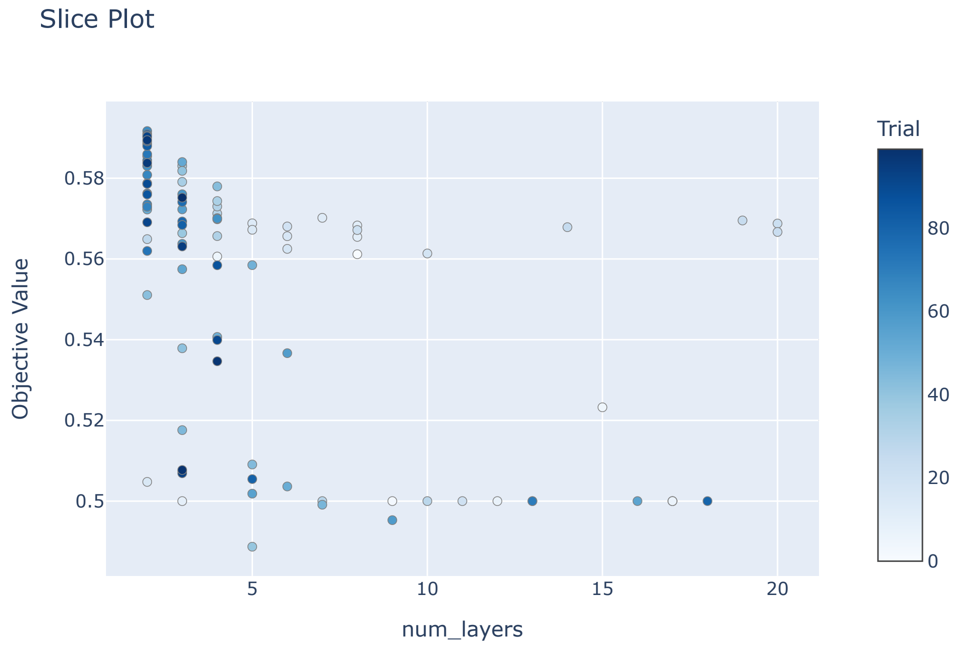

Figure 6.26: Slice plot showing the impact the hyperparameter "num_layers" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

Figure 6.27 provides a visual representation of the significance of the different activation functions. The results demonstrates that the logistic, also known as the sigmoid, activation function gives the best results.

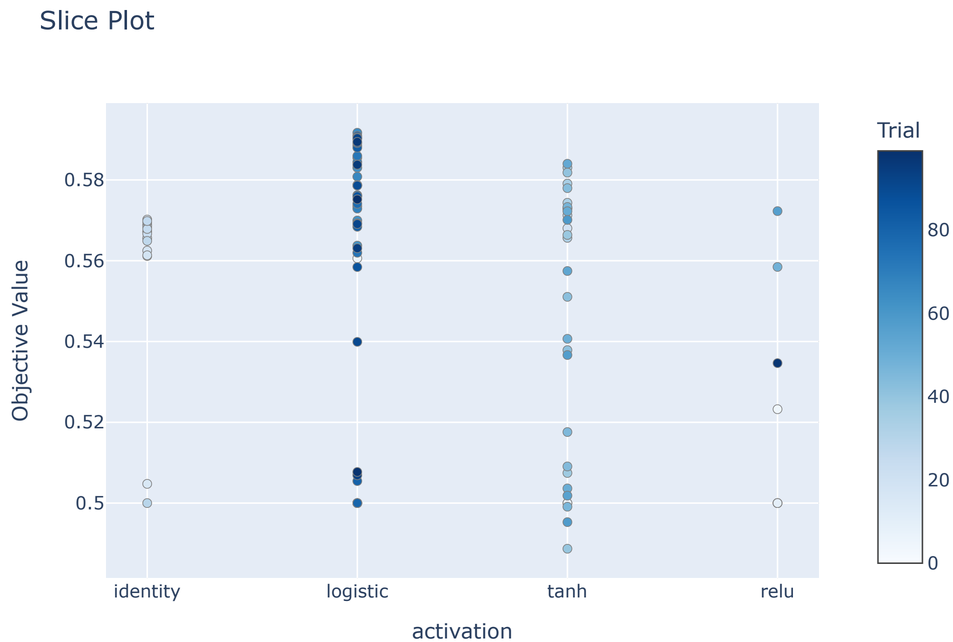

Figure 6.27: Slice plot showing the impact the hyperparameter "activation" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

6.4.2 Second Results

To further investigate the importance of the hyperparameters, a second Bayesian optimization has been done with 50 iterations. Table 6.15 displays the hyperparameters that has been chosen for further investigation with their respective domain. "num_layers" is chosen because it could be interesting to investigate if 1 layer might give better results than more layers, and also to check if there is a clear tendency that more layers give poorer results. "neurons_per_layer" and "max_iter" are chosen because they are the second and third most influential continuous hyperparameters.

Table 6.15: The hyperparameters with their respective search domain for the second deep learning hyperparameter tuning.

Hyperparameter	Search Domain
num_layers	$[1, 5]$
neurons_per_layer	$[100, 300]$
max_iter	$[2, 100]$

Table 6.16 shows the outcomes of the hyperparameter tuning process. The second hyperparameter tuning took 46 hours and 10 minutes to execute and yielded an AUC value of 0.5923. This means that there was an improvement of 0.0006 in the objective value for the second optimization compared to the first one.

Table 6.16: The hyperparameters with their optimal values for the second deep learning hyperparameter tuning.

Hyperparameter	Value
num_layers	2
neurons_per_layer	274
max_iter	85

Figure 6.28 shows a history plot of the second Bayesian optimization. It is evident that the best objective value shows smaller improvements in the beginning, and then eventually slows down.

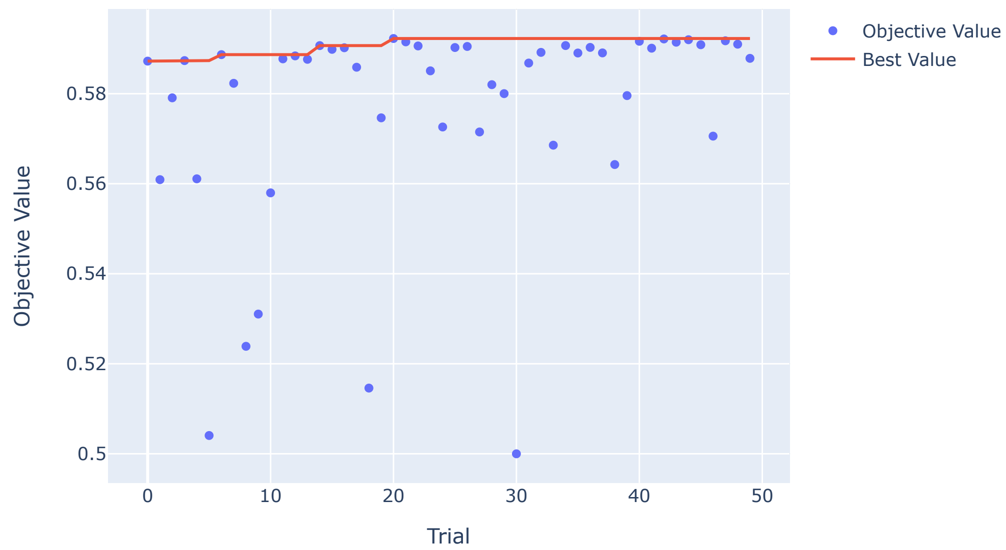

Figure 6.28: Optimization history plot of the Bayesian optimization. The objective values (AUC) for each iteration are denoted by blue dots. The red line corresponds to the best objective value obtained so far.

Figure 6.29 shows the degree of importance that each hyperparameter has on the objective value. The plot indicates that "num_layers" is the most significant hyperparameter. "max_iter" and "neurons_per_layer" appear to have a small impact on the objective value.

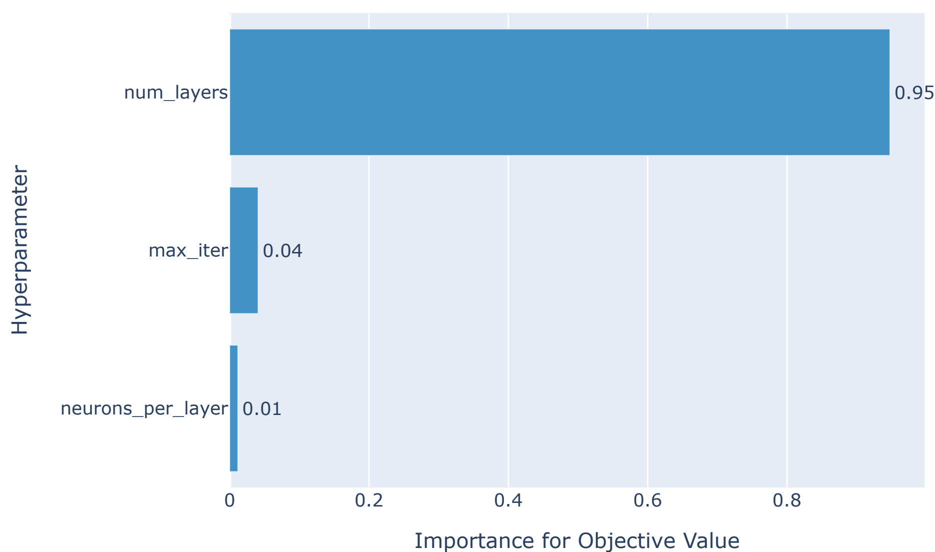

Figure 6.29: Hyperparameter importance plot indicating the importance each hyperparameter has on the objective value.

Figure 6.30 provides a visual representation of the impact of the amount of layers in the neural

network. The results demonstrate that one to two layers gives the best results.

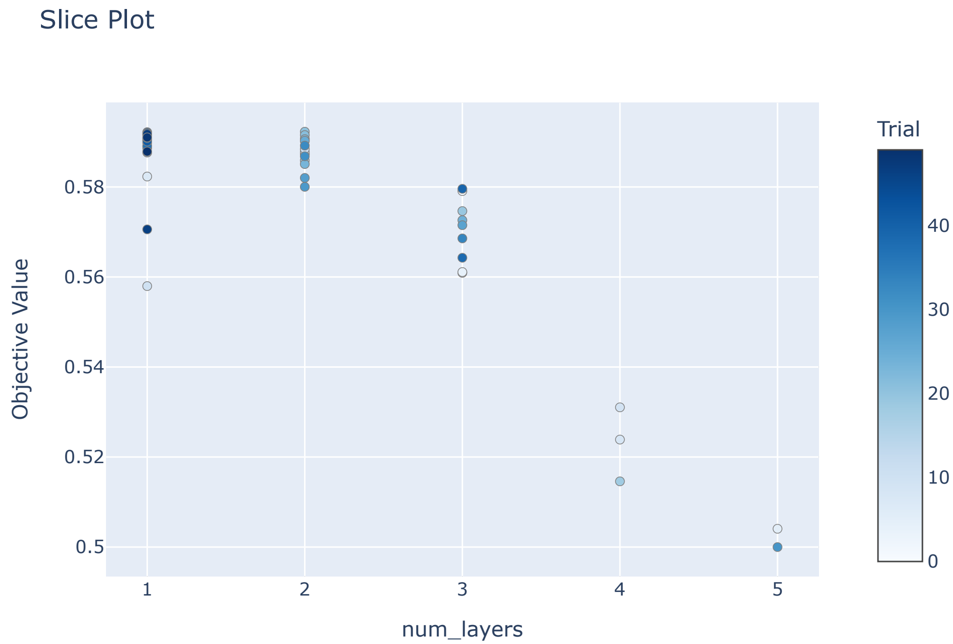

Figure 6.30: Slice plot showing the impact the hyperparameter "num_layers" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

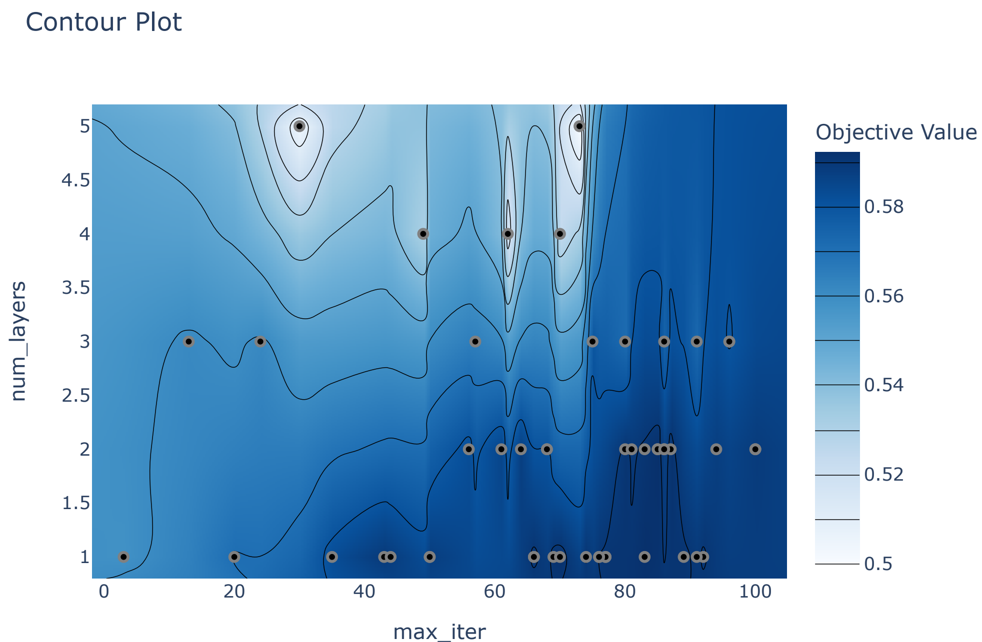

Figure 6.31: Contour plot showing how different combinations of "num_layers" and "max_iter" changes the objective value. Darker color represents better objective values.

Figure 6.31 shows a Contour plot. In this plot one can see how two different hyperparameters influence the objective value. Darker color represents better objective values. It is clear that both "num_layers" and "max_iter" plays an important role on the model performance.

Outprint from the code as well as the slice plots for the other hyperparameters are included in Appendix E.

6.4.3 Threshold

The deep learning model with optimal hyperparameters achieved its best performance at a threshold of 0.062. Figure 6.32 illustrates the impact of varying the threshold between zero and one on the objective value defined in Chapter 5. It is evident from the graph that there is a distinct optimal threshold that maximizes the model's performance.

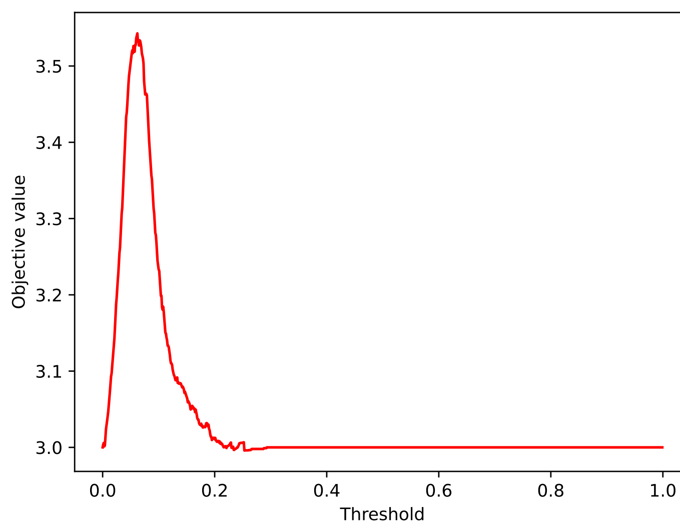

Figure 6.32: A plot showing how different values of the threshold affect an objective value.

6.5 Comparing the Tuned Models

The predictive models were all trained using the optimal hyperparameters and optimal threshold selected in the sections above. This training was carried out exclusively on the training set, after which the predictive models were tested using the test set. The classification results of each predictive model on the test set are displayed in Table 6.17, 6.18, and 6.19, in the form of confusion matrices.

The three model predictions shows somewhat similar results. A thing to notice is that logistic regression tends to classify customers as becoming active a bit more frequently than the other models.

Table 6.17: Confusion matrix from logistic regression. The training of the model was performed on the training set using optimal hyperparameters, and the model was evaluated on the test set using a optimal threshold of 0.056. 0 represents remaining passive while 1 represents becoming active.

True \ Predicted	0	1
0	20960	40536
1	1046	3152

Table 6.18: Confusion matrix from gradient boosted decision trees. The training of the model was performed on the training set using optimal hyperparameters, and the model was evaluated on the test set using a optimal threshold of 0.060. 0 represents remaining passive while 1 represents becoming active.

True \ Predicted	0	1
0	32278	29218
1	1182	3016

Table 6.19: Confusion matrix from deep learning. The training of the model was performed on the training set using optimal hyperparameters, and the model was evaluated on the test set using a optimal threshold of 0.062. 0 represents remaining passive while 1 represents becoming active.

True \ Predicted	0	1
0	30760	30736
1	1524	2674

Table 6.20 displays the results of each classification metric. It is important to note that the Brier score is the only metric where minimization is the goal. Consequently, to transform the objective of all metrics presented in the table into maximization, 1−Brier score is employed.

Gradient boosted decision trees still outperforms both logistic regression and deep learning for every all-round metric. The only metric where some of the other predictive models shows better results are in specificity. Nonetheless, specificity alone does not provide a complete picture of the overall prediction performance.

Logistic regression and deep learning still performs somewhat similar in all the all-round metrics. Nonetheless, deep learning does a better predicting job.

Table 6.20: Results from classification metrics on the test set for all predicting models with optimal hyperparameters and threshold.

Predicting model	AUC	1−Brier score	MCC	BACC	Accuracy	Sensitivity	Specificity
Logistic regression	0.5728	0.9404	0.0475	0.5458	0.3670	0.3408	0.7508
Gradient boosted decision trees	0.6606	0.9407	0.1190	0.6217	0.5372	0.5249	0.7184
Deep learning	0.5967	0.9406	0.0671	0.5686	0.5089	0.5002	0.6370

6.5.1 The tuning process

Table 6.21 displays information on different aspects of the tuning process. It is evident that the runtime varies significantly among the predictive models. The first Bayesian optimization required the training of 300 predictive models, whereas the second required the training of 150. Consequently, the runtime can be considerably high.

Training a logistic regression model usually requires minimal computational power, therefore, the runtime for training a single model is generally quite low. This is particularly evident in the second optimization. However, during the first optimization, the runtime for training a single model was at times considerably extended because an inefficient "solver" was utilized to identify the coefficients β_j. This solver was not employed during the second Bayesian optimization, resulting in a more efficient runtime. It is worth noting that deep learning, on the other hand, is a time-intensive process.

Upon examining the optimal AUC scores for each optimization, it becomes evident that the predictive models did not experience significant improvement from the first Bayesian optimization to

the second. In particular, there was no improvement in the performance of the logistic regression model. Moreover, one can see that gradient boosted decision trees outperforms the other two models.

Table 6.21: Information regarding the tuning process for each predicting model. The best objective value and the runtime is displayed for both the first and the second Bayesian optimization

Predicting model	First runtime	Best AUC (first)	Second runtime	Best AUC (second)
Logistic regression	10 hours and 36 minutes	0.5697	2 hours and 59 minutes	0.5697
Gradient boosted decision trees	8 hours and 6 minutes	0.6569	11 hours and 28 minutes	0.6592
Deep learning	53 hours and 6 minutes	0.5917	46 hours and 10 minutes	0.5923

Table 6.22 presents a comparison of the classification metrics for all models before and after tuning. All models were trained on the same training set and evaluated using the same test set. A positive value indicates an improvement in the corresponding metric, whereas a negative value indicates an reduction. Overall, deep learning demonstrated the most significant improvement, while gradient boosted decision trees saw modest improvement across most metrics. In contrast, the performance of the logistic regression model slightly declined. Another thing to notice is that all the predicting models saw a decline in both accuracy and sensitivity. This is likely due to the changing to a smaller threshold value for every model.

Table 6.22: The difference in the results from classification metrics on all models before and after tuning.

Predicting model	AUC	1−Brier score	MCC	BACC	Accuracy	Sensitivity	Specificity
Logistic regression	−0.0003	0.0001	−0.0055	−0.0077	−0.2147	−0.2451	0.2296
Gradient boosted decision trees	0.0145	0.0000	0.0174	0.0179	−0.0039	−0.0071	0.0428
Deep learning	0.0387	0.0104	0.0221	0.0278	−0.20009	−0.2344	0.2899

6.5.2 Investigating threshold importance

Figure 6.33, 6.34, and 6.35 visualizes how the threshold influences different metrics. As before, these figures indicates that the MCC is slightly less sensitive to the threshold in comparison to the BACC. Moreover, it becomes evident that the accuracy metric is still not an ideal test for unbalanced datasets.

The ideal threshold for each predictive model remains relatively consistent across all metrics except for the accuracy.

It is noteworthy that even though the maximum MCC and BACC scores for the deep learning model are not as high as those for the gradient boosted decision trees model, the scores of deep learning appear to be less affected by changes in the threshold value. The gradient boosted decision trees model also exhibits a notable behavior with respect to both the MCC and the BACC metrics, as certain threshold values yield outcomes that perform even worse than a random classifier.

Appendix F contains similar graphs, although these focus on Sensitivity and Specificity instead.

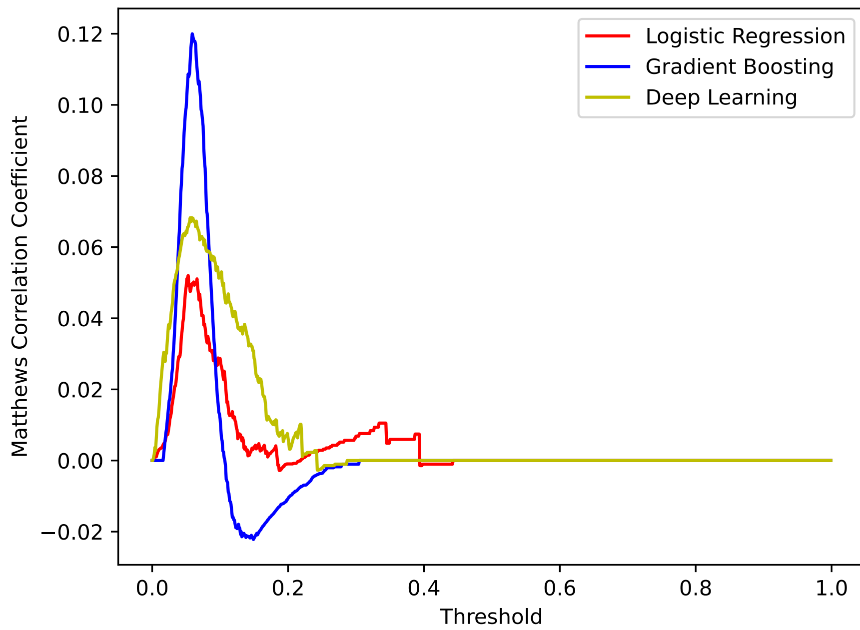

Figure 6.33: A plot illustrating the impact of various threshold values on the Matthews Correlation Coefficient (MCC) for different predictive models with optimal hyperparameters. Logistic regression is displayed in red, gradient boosted decision trees is displayed in blue and deep learning is displayed in yellow.

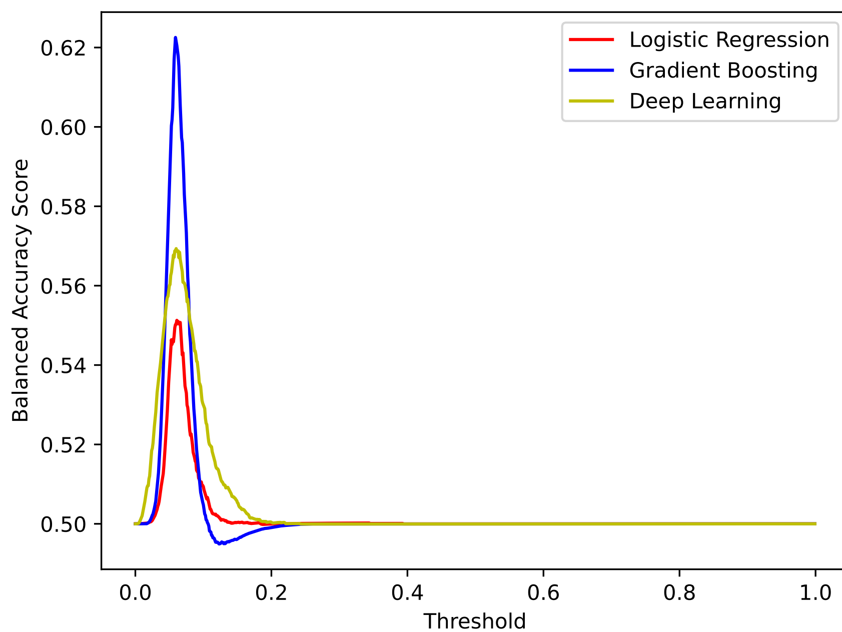

Figure 6.34: A plot illustrating the impact of various threshold values on the Balanced Accuracy (BACC) for different predictive models with optimal hyperparameters. Logistic regression is displayed in red, gradient boosted decision trees is displayed in blue and deep learning is displayed in yellow.

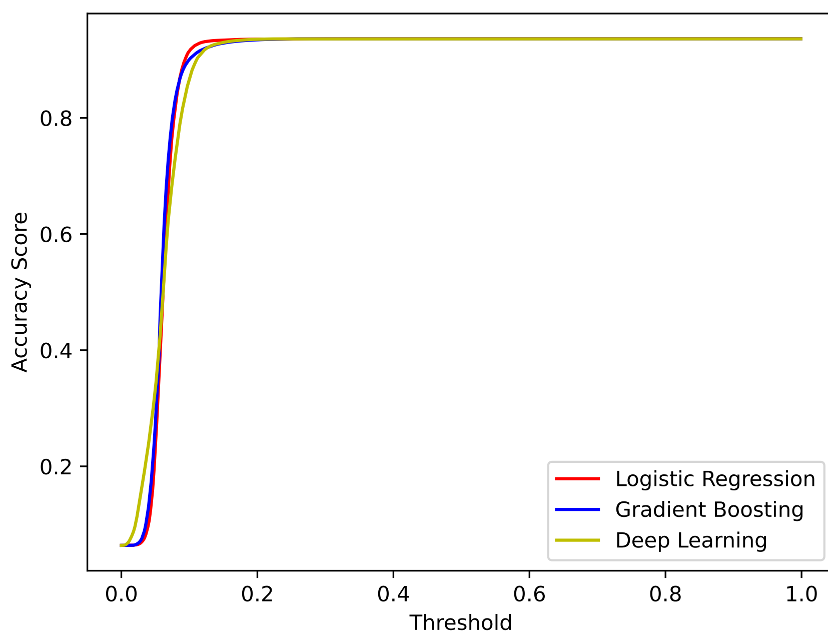

Figure 6.35: A plot illustrating the impact of various threshold values on the Accuracy for different predictive models with optimal hyperparameters. Logistic regression is displayed in red, gradient boosted decision trees is displayed in blue and deep learning is displayed in yellow.

6.5.3 Feature importance

Figure 6.36 and 6.37 exhibit the feature importance for each predictive model. These graphs provide insight into the importance of each feature in the respective models. The feature importance for logistic regression is determined by the absolute value of the coefficients β_j. The scale is set so that the feature with the highest coefficient has a feature importance of 100.

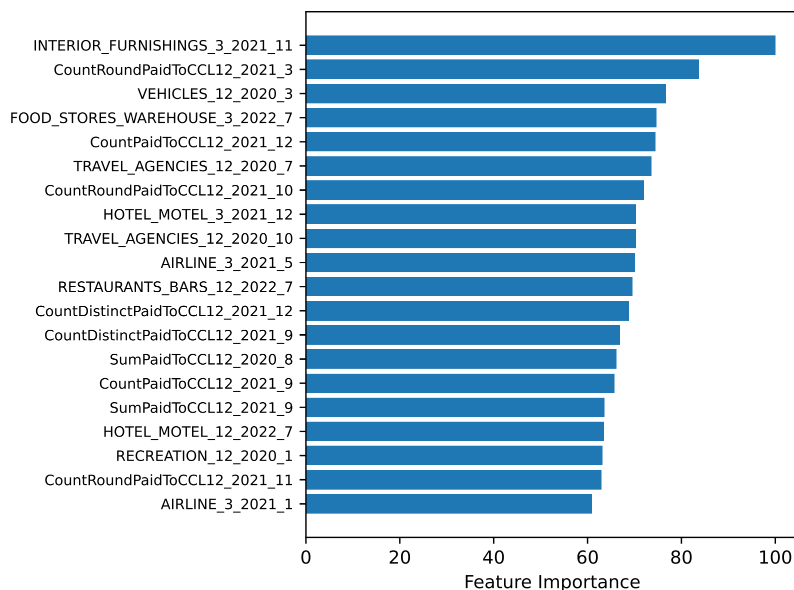

Figure 6.36: Feature importance for logistic regression with optimal hyperparameters. The top 20 most important features are displayed.

The feature importance regarding gradient boosted decision trees is still evaluated based on the improvement of the performance measure attributed to each split in a tree. This is weighted by the number of observations for which each node is responsible for. The overall feature importance is determined by averaging the feature importance across all the trees in the model.

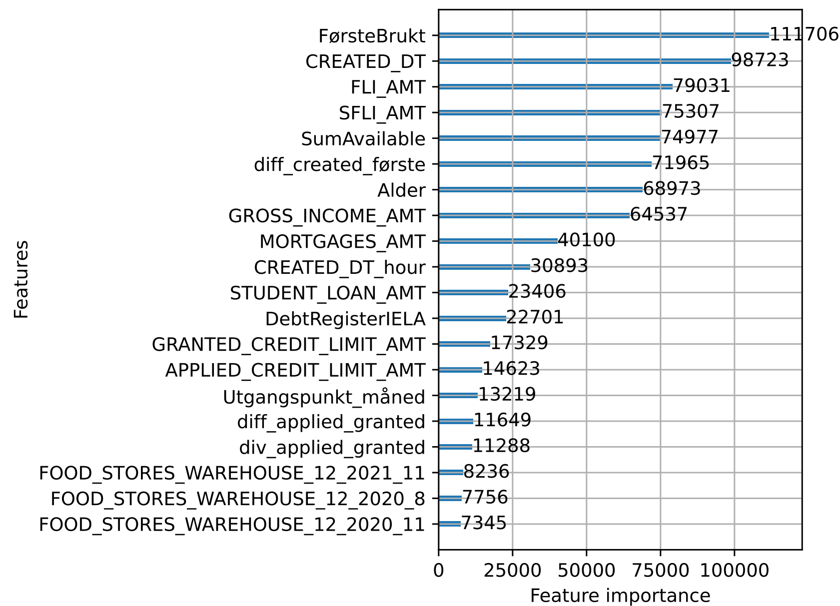

Figure 6.37: Feature importance for gradient boosted decision trees with optimal hyperparameters. The top 20 most important features are displayed.

One can see that logistic regression and gradient boosted decision trees do not share any of the top 20 most important features. It also becomes evident that the top 20 most important features for logistic regression now only contain historical credit card uses or transactions.

One thing to notice is that "INTERIOR_FURNISHINGS_3_2021_11" and "CountRoundPaidToCCL12_2021_3" are still the top two most important features for logistic regression. Whereas "FørsteBrukt" is still the most important feature for gradient boosted decision trees, but "CREATED_DT" has now taken the second place.

The feature importance for deep learning is not provided in the Scikit-learn package, there will as a result be looked at SHAP values for deep learning. This will be done in the last section of this chapter. The SHAP values for the other predicting models will also be included.

Outprint from the code as well as threshold plots for the tuned models are for included in Appendix F.

6.6 Feature Importance through SHAP values

In this section, there will be employed SHAP values to explore feature importance across different predictive models.

Obtaining SHAP values for the default predictive models required some runtime, but eventually, everything fell into place. However, when it came to the tuned predictive models, the process did not yield the desired results. Finding SHAP values from the tuned logistic regression model followed essentially the same procedure as for the default logistic regression model with the desired results.

The expected execution time for finding SHAP values for the tuned gradient boosted decision trees

model, however, was estimated to be 294 hours, this is a little over 12 days. Due to the excessive duration, the code was interrupted.

The process for the tuned deep learning model appeared to be functioning well until the the SHAP value plots were displayed. As showed in Figure F.7, both plots turned out to be completely blank, without any information. The exact reason behind this remains unknown, but one possibility is that a neural network designed for a dataset with 1200 features, incorporating 2 hidden layers with 274 neurons per layer, could potentially consume excessive memory when computing the SHAP values. Consequently, the computer might have released some of this memory, resulting in the blank plots, this is however unknown. The process of generating SHAP values for the tuned deep learning model was attempted twice, but produced the same outcome each time.

Table 6.23 shows the execution time for getting the SHAP values for the different predictive models, both with and without optimized hyperparameters. It is important to know that the execution time for getting the SHAP values for the tuned gradient boosted decision trees model is just an estimation based on the first iterations, however, based on the estimations of the execution time for the other predicting models, this estimate should be fairly accurate.

Table 6.23: The execution time for getting the SHAP values for the different predictive models, both with and without optimized hyperparameters

Predicting model	Runtime Default	Runtime Tuned
Logistic regression	26 hours and 40 minutes	26 hours and 56 minutes
Gradient boosted decision trees	28 hours and 19 minutes	294 hours
Deep learning	42 hours and 34 minutes	36 hours and 51 minutes

6.6.1 Default predicting models

The SHAP values for the default predicting models are showed in Figure 6.38, 6.39 and 6.40, one can see that the predicting models feature importance varies a lot across models. Logistic regression ranks the historical credit card features as the most important features with "OTHER_RETAIL_12_2021_2" being the most important feature. Gradient boosted decision trees also has a lot of historical credit card features as the most important features with "SumPaidToCCL12_2021_11" as the most important feature, this feature comes from the historical transactions dataset. Deep learning on the other hand sets personal features from both the appliance dataset and the fundamental dataset as being the most important features, with "ProductId_8" as being the most important feature.

Additionally, for both deep learning and gradient boosted decision trees, it seems that "Utgangspunkt_måned" and "FørsteBrukt" are important features.

It should be noted that the earlier feature importance plots in this chapter are a direct consequence of the parameters of the predicting models. An example of this is how the feature importance for logistic regression is determined by the absolute value of the coefficients β_j only. However, the SHAP values from the different predicting models are not a direct consequence of these parameters, but rather the evaluation of the significance of each feature by measuring its impact on the model's prediction given all the other features.

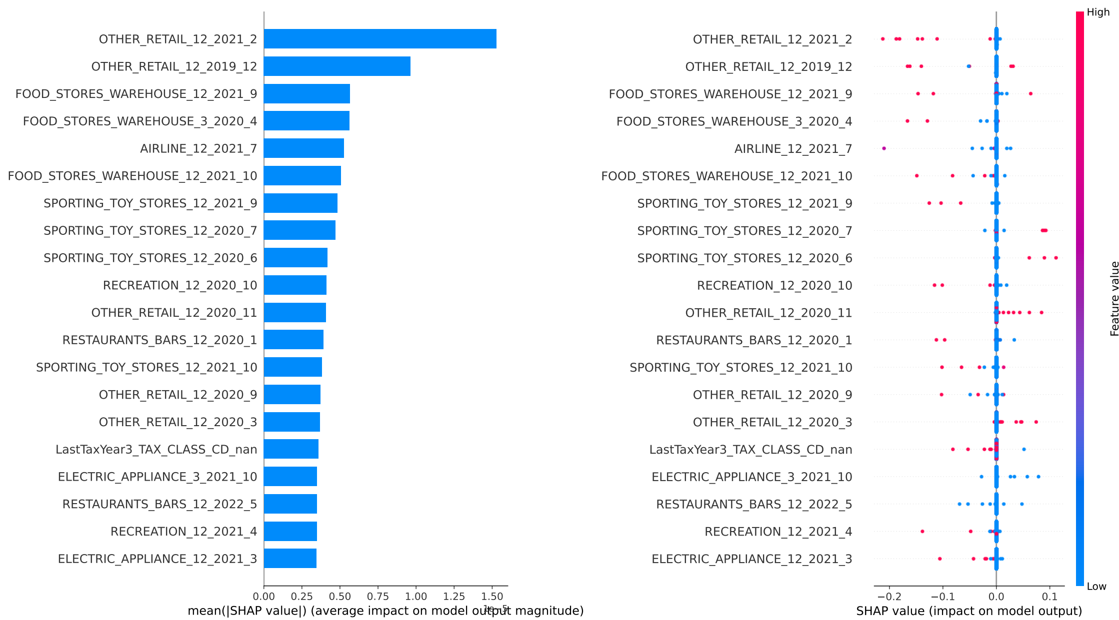

Figure 6.38: A figure showcasing the SHAP values for the default logistic regression model. The left graph shows the mean of the absolute value for all the SHAP values in each feature, hence showcasing the average impact each feature has on the model output. The right graph shows the densities of how the SHAP values contribute to the model output for each feature. This graph also shows how high and low feature values contribute to the model output with red denoting a high feature value and blue denoting a low feature value.

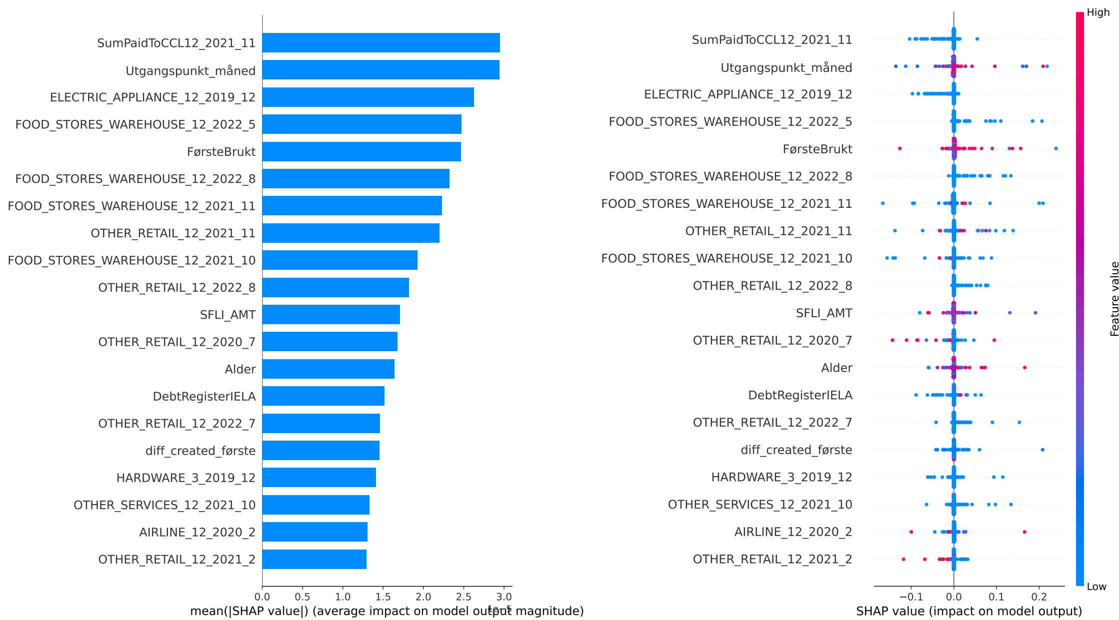

Figure 6.39: A figure showcasing the SHAP values for the default gradient boosted decision trees model. The left graph shows the mean of the absolute value for all the SHAP values in each feature, hence showcasing the average impact each feature has on the model output. The right graph shows the densities of how the SHAP values contribute to the model output for each feature. This graph also shows how high and low feature values contribute to the model output with red denoting a high feature value and blue denoting a low feature value.

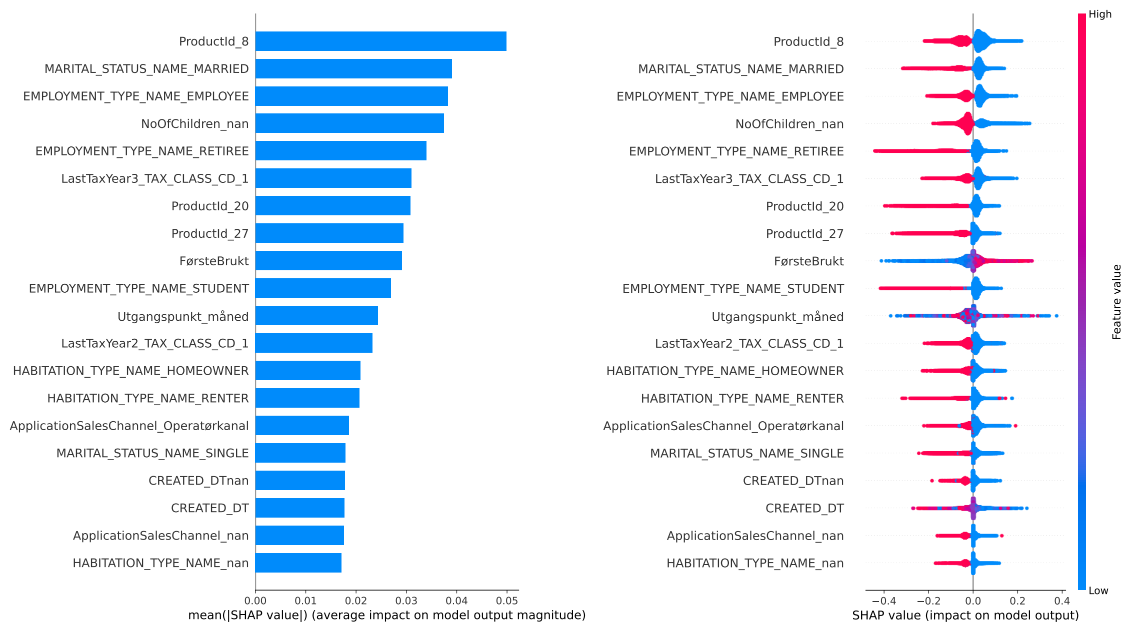

Figure 6.40: A figure showcasing the SHAP values for the default deep learning model. The left graph shows the mean of the absolute value for all the SHAP values in each feature, hence showcasing the average impact each feature has on the model output. The right graph shows the densities of how the SHAP values contribute to the model output for each feature. This graph also shows how high and low feature values contribute to the model output with red denoting a high feature value and blue denoting a low feature value.

6.6.2 Tuned predicting models

In Figure 6.41 one can see the SHAP values for the tuned logistic regression model. It becomes evident that there is not any major change in the feature importance after tuning, and "OTHER_RETAIL_12_2021_2" is still the most important hyperparameter for both the default and the tuned logistic regression model.

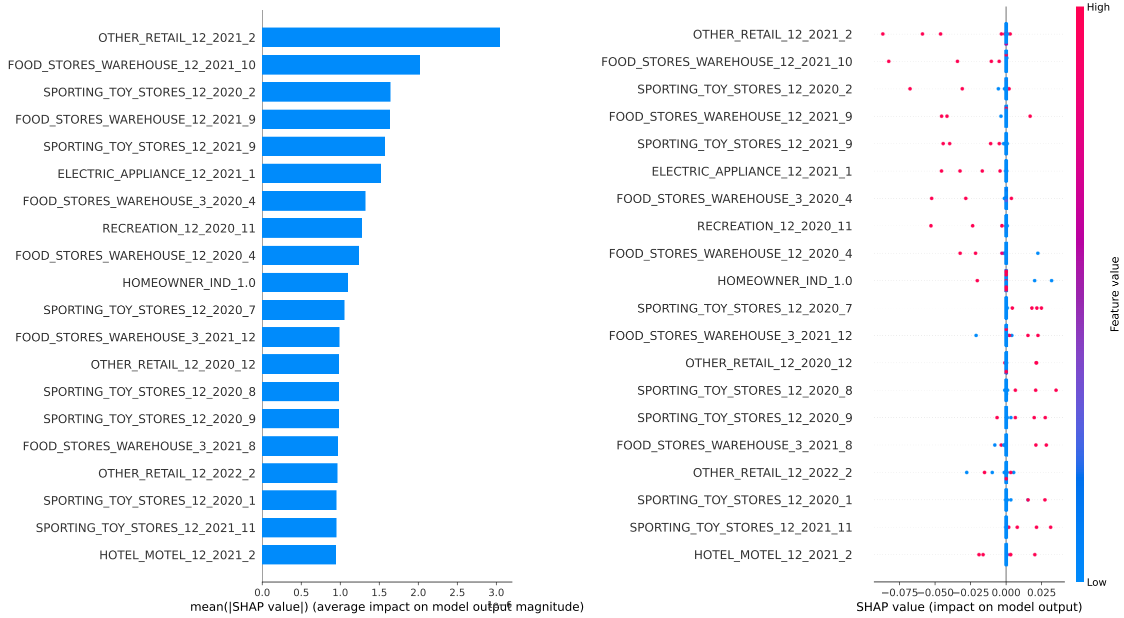

Figure 6.41: A figure showcasing the SHAP values for the tuned logistic regression model. The left graph shows the mean of the absolute value for all the SHAP values in each feature, hence showcasing the average impact each feature has on the model output. The right graph shows the densities of how the SHAP values contribute to the model output for each feature. This graph also shows how high and low feature values contribute to the model output with red denoting a high feature value and blue denoting a low feature value.

Outprints for the execution time for finding the SHAP values along with the anticipated execution time for the tuned gradient boosted decision trees model are included in Appendix F.

Chapter 7

Discussion

SpareBank1 is distributing credit cards to customers, however some customers who possess credit cards are not using them. The objective of this thesis was to develop and refine models capable of classifying passive customers into two categories: those who will remain passive for a given month and those who will become active for a given month. The training process involved using information from four distinct datasets. A comparative analysis of three binary classification models; logistic regression, gradient boosted decision trees, and deep learning, was conducted using the provided data from Sparebank1. The hyperparameters for each classification model were optimized through Bayesian optimization, aiming to maximize an objective value. The outcome of this optimization process resulted in performance improvements for two out of the three classification models.

Initially, the classification models underwent training using default hyperparameters. The models were then evaluated on the test set, using a threshold of 0.064. Various classification metrics were employed, and the corresponding outcomes are presented in Table 6.4. Gradient boosted decision trees saw superior predicting performance across all all-round metrics. Logistic regression attained the second-best results, while deep learning yielded the least favorable outcomes.

Upon closer examination, the threshold value of 0.064 did seem as a potentially optimal threshold. Additionally, the investigation of feature importance revealed difference between logistic regression and gradient boosted decision trees. In the case of logistic regression, features from the historical credit card dataset was being identified as the most significant. This trend likely arises due to the calculation of feature importance for logistic regression. Features with a limited number of outliers, where these outliers appear to contribute significantly, or in other words, where the coefficients β_j have large absolute values, seem to be prioritized. Unfortunately, the deep learning package used did not provide support for feature importance, so the feature importance for deep learning had to be investigated later when using SHAP values.

Using Bayesian optimization did not only lead to the discovery of optimal hyperparameter values, but also provided valuable insights into the significance and impact of these hyperparameters on the objective value. When applied to logistic regression, Bayesian optimization revealed that the hyperparameters had a relatively lower influence on the objective value compared to the other predictive models. Furthermore, a notable finding was the finding of a clear optimum for the L2 penalization. Additionally, it was observed that using a second, more refined optimization did not yield any improvements in the objective value.

By employing Bayesian optimization to gradient boosted decision trees, it became evident that the hyperparameters had a notable impact on the objective value. It was observed that increasing the number of trees and reducing the learning rate enhanced the predictive performance of the model. Moreover, the analysis indicated that using a second, more refined optimization, led to improvements in the objective value. Additionally, it was discovered that the model had a preference for a substantially lower learning rate compared to the default value.

The usage of Bayesian optimization on deep learning showed a substantial influence of the hyperparameters on the objective value. Certain combinations of hyperparameters resulted in an AUC over 0.59, while others yielded outcomes worse than those of a random classifier. Furthermore, it became evident that the model exhibited a preference for two hidden layers in the neural network. The introduction of a second optimization demonstrated more clearly that incorporating three or more hidden layers resulted in a significant decline in predictive performance. Moreover, the objective value in the second optimization saw a slight improvement.

The threshold optimization showed that every predicting model preferred a threshold around 0.06, which is really close the the fraction of positive cases in the dataset.

All the classification models were then trained using tuned hyperparameters. The models were then evaluated on the test set, using optimized thresholds. Various classification metrics were employed, and the corresponding outcomes are presented in Table 6.20. Gradient boosted decision trees maintained its superiority over the other models across the all-round metrics. Notably, deep learning emerged as the second-best performer, while logistic regression yielded the least favorable results. These findings were consistent with expectations, considering the objective value observed during the hyperparameter tuning process.

Table 6.22 shows the improvements achieved by each predictive model in each binary classification metric following the hyperparameter tuning process. In general, logistic regression exhibited a slight decrease in performance, and the exact reason behind this outcome remains uncertain. It is evident that the hyperparameters in the logistic regression model had minimal impact on the objective value. Since default hyperparameters are typically robust, the tuning process might not have contributed significantly, and the slight decline in performance could be attributed to chance. On the other hand, deep learning experienced a substantial performance boost after tuning. Lastly, gradient boosted decision trees showed slight improvements in the all-round metrics following the tuning process.

The tuning process itself involves training numerous models. In this case, Bayesian optimization was conducted with 100 and 50 iterations. This was done with cross-validation where $N = 3$. As a result, a total of 450 models were trained for each predictive model. Table 6.21 illustrates the variations in runtime for the tuning process across the different predictive models. Logistic regression, being a simple model, requires relatively little computational resources. On the other hand, deep learning heavily relies on computational power, as evidenced by its longer runtime during the tuning process. Gradient boosted decision trees falls in between these two extremes. Although it is based on a computationally intensive algorithm, it incorporates a highly optimized histogram-based decision tree learning approach, providing significant efficiency and memory consumption advantages.

Upon examining the feature importance in the tuned models, it is apparent that there is not a significant difference before and after the tuning process. Gradient boosted decision trees and logistic regression continue to make more or less different feature importance plots. Notably, "FørsteBrukt" remains the most important feature for gradient boosted decision trees, while "INTERIOR_FURNISHINGS_3_2021_11" remains the most important feature for logistic regression.

The creation of SHAP values was successfully performed for the default predicting models. However, when it came to the tuned predicting models, certain issues arose during the process.

The SHAP values for the default prediction models showed significant variations in the feature importance across different models. Logistic regression still ranks the historical credit card features as the most important, with "OTHER_RETAIL_12_2021_2" being the top-ranked feature. Similarly, gradient boosted decision trees assigns high importance to many historical credit card features, however the most important features are features from all the datasets, with "SumPaidToCCL12_2021_11" standing out as the most influential feature, which is from the historical transactions dataset. On the contrary, deep learning prioritizes personal features from both the appliance dataset and the fundamental dataset as being the most important, with "ProductId_8" emerging as the most significant feature.

Upon observation, it becomes apparent that the most important features for logistic regression

correspond to features with SHAP values that have a few outliers with significant high absolute values, while the remaining SHAP values have low or zero absolute values. A similar pattern can be observed for gradient boosted decision trees, nevertheless there is a notable increase in the presence of outliers with high SHAP values. However, in the case of deep learning, this trend does not hold true. Instead, each observation in each feature seems to receive a non-zero SHAP value, indicating that the contribution of each observation is rarely negligible. While the SHAP values provides useful insights to feature importance, extracting the SHAP values can be a very computational heavy task.

After data pre-processing, hyperparameter tuning, and threshold optimization, it remains challenging to create a single model that accurately predicts which type of passive customers that will become active at what given month. One can argue that the data provided by Sparebank1 may not have been sufficiently informative to achieve precise predictions. Although the dataset was not sparse, there were missing values in several explanatory variables. Additionally, it is important to recognize that the prediction task at hand is challenging. It involves not only predicting whether a customer will become active, but also at what month this transition might occur.

Nevertheless, despite these challenges, the predictive models developed did perform significantly better than random classifiers, indicating that the data from Sparebank1 contains useful information.

For this dataset, there was a notable difference in the performance of the three predicting models. The gradient boosted decision trees model saw much better predictive capabilities compared to both logistic regression and deep learning. Deep learning, although computationally heavy and requiring careful tuning for satisfactory results, outperformed logistic regression. However, deep learning may not be a suitable option for this specific task due to it falling short in every way compared to gradient boosted decision trees. It lacks model interpretability completely, it is more computationally heavy, and it produces inferior results.

Therefore, the choice for a predictive model for this kind of task is really between logistic regression and gradient boosted decision trees. This choice depends on the specific task and desired outcome. If the objective is to achieve the most precise predictions for a given task, implementing gradient boosted decision trees and using Bayesian optimization for tuning hyperparameters, would be a worthwhile approach. On the other hand, if the task primarily requires simple predictions, a logistic regression model without optimized hyperparameters may be sufficient.

When selecting the most appropriate model for a specific dataset and task, it is essential to consider the trade-off between model complexity, interpretability, computational cost, and the desired level of predictive performance.

In the context of response modeling, the balance between interpretability and model performance is important. One crucial aspect of model building is gaining insights into how certain features contribute to the response variable. In terms of interpretability and simplicity, logistic regression offers a straightforward approach. It allows for understanding the impact of each explanatory variable on the response. On the other hand, gradient boosted decision trees is a complex model that lacks such interpretability. Therefore, if the priority is to conduct a statistical analysis where interpretability is crucial, logistic regression would likely be the preferred method.

All i all, it is crucial to carefully assess the specific requirements of the task at hand, such as the need for interpretability and the level of prediction accuracy desired. By considering these factors, one can make an informed decision about the most suitable predicting model.

Chapter 8

Conclusion

The main objective of this thesis was to predict what type of passive credit card users that would become active credit card users, and in what month this transition would happen. The developed predictive models demonstrated significant better predictive performance than a random classifier. The predictive models also provided useful information on how certain features contributed to the response. However, none of the models achieved accurate classification of customers. It is important to recognize the complexity associated with predicting individual behavior, which is influenced by various factors beyond the scope of datasets, and although the dataset provided was informative, the predictive task proved to be difficult.

This thesis has demonstrated that more complex classification models, such as gradient boosted decision trees and deep learning, can outperform a simpler model like logistic regression. Optimizing the hyperparameters of gradient boosted decision trees and deep learning resulted in a improved classification performance of these models. The degree of improvement varied, with deep learning showing a significant improvment, while gradient boosted decision trees showed a more modest improvement. On the other hand, tuning the hyperparameters of logistic regression did not result in any improvement in classification performance.

The most important hyperparameter for logistic regression was determined to be "solver", whereas for gradient boosted decision trees this was "learning_rate". As for deep learning, the most important hyperparameter was "num_layers." The tuning process did not significantly affect the feature importance. Examining the SHAP values, it was found that "OTHER_RETAIL_12_2021_2" emerged as the most important feature for logistic regression, "SumPaidToCCL12_2021_11" for gradient boosted decision trees, and "ProductId_8" for deep learning.

8.1 Future Work

There are several suggestions for future work. First of all, it is not totally clear how each of the datasets provided by Sparebank1 contributed to the predictive performance of the different models. Trying out different combinations of datasets, and see the predicting results might give even more insight in what type of data that is most useful in this predicting task. Furthermore, finding more data, and see if the predicting results would improve, could also be interesting. Another task that would have been interesting to look at, is trying to predict only if a customer will become active or not, and thus not focus on the month that they become active in, and thus see if that would improve the predictive performance of the models, and in that case, how much. The response variable in the dataset is also very imbalanced, and there are several methods to make the data more balanced. For example, one could see if both undersampling or oversampling could improve the predictive performance.

Runtime was a major factor in this task, so an idea would be to look at ways to remove less

important features. This should be interesting in two ways; how the runtime changes as features are removed, and how the predicting results changes as features are removed. Another scenario is that if more computing power was available, it would then be interesting to see if there could have been achieved better results by exploring more hyperparameters with broader domains through more iterations with Bayesian optimization. A further investigation of the SHAP values would also then be possible.

Lastly, it could be interesting to see how other predictive models would perform on this dataset, and also if combining different predictive models could lead to better performance.

Bibliography

[1] Arun Addagatla. 'Maximum Likelihood Estimation in Logistic Regression'. In: *Medium* (2021). URL: https://arunaddagatla.medium.com/maximum-likelihood-estimation-in-logistic-regression-f86ff1627b6.

[2] Avcontentteam. 'Tree Based Algorithms: A Complete Tutorial from Scratch (in R & Python)'. In: *Analytics Vidhya* (2021). URL: https://www.analyticsvidhya.com/blog/2016/04/tree-based-algorithms-complete-tutorial-scratch-in-python/.

[3] Jason Brownlee. 'A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise'. In: *Machine Learning Mastery* (2021). URL: https://machinelearningmastery.com/threshold-moving-for-imbalanced-classification/.

[4] Jason Brownlee. 'Essence of Bootstrap Aggregation Ensembles'. In: *Machine Learning Mastery* (2021). URL: https://machinelearningmastery.com/essence-of-bootstrap-aggregation-ensembles/.

[5] Quan Yuan Chao Yang Mingyang Chen. 'The application of XGBoost and SHAP to examining the factors in freight truck-related crashes: An exploratory analysis'. In: (2021). URL: https://www.sciencedirect.com/science/article/abs/pii/S0001457521001846?via%3Dihub.

[6] Krishnendu Chaudhury. 'Math and Architectures of Deep Learning Version 10'. In: (2022).

[7] Microsoft Corporation. *Experiments*. 2020. URL: https://lightgbm.readthedocs.io/en/v3.3.2/Experiments.html.

[8] Microsoft Corporation. 'LightGBM's documentation'. In: *LightBGM-readthedocs* (2022). URL: https://lightgbm.readthedocs.io/en/v3.3.3/index.html.

[9] Microsoft Corporation. 'Parameters'. In: *LightBGM-readthedocs* (2022). URL: https://lightgbm.readthedocs.io/en/latest/Parameters.html.

[10] Author Jakub Czakon. '24 Evaluation Metrics for Binary Classification And When to Use Them'. In: *MLOps Blog* (2022). URL: https://neptune.ai/blog/evaluation-metrics-binary-classification.

[11] Jia Wu — Xiu-Yun Chen — Hao Zhang — Li-Dong Xiong — Hang Lei — Si-Hao Deng. 'Hyperparameter Optimization for Machine Learning Models Based on Bayesian Optimization'. In: *KeAi* (2019).

[12] scikit-learn developers. 'sklearn.neural_network.MLPClassifier'. In: (2023). URL: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.

[13] Scikit-learn developers. 'Cross-validation: evaluating estimator performance'. In: *Scikit-learn* (2022). URL: https://scikit-learn.org/stable/modules/cross_validation.html.

[14] Scikit-learn developers. 'sklearn.linear_model.LogisticRegression'. In: *Scikit-learn* (2022). URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression.

[15] XGBoost developers. 'Introduction to Boosted Trees'. In: *XGBoost Documentation* (2022). URL: https://xgboost.readthedocs.io/en/stable/tutorials/model.html.

[16] XGBoost developers. 'XGBoost Documentation'. In: *XGBoost Documentation* (2022). URL: https://xgboost.readthedocs.io/en/stable/index.html.

[17] IBM Cloud Education. 'Neural Networks'. In: *IBM Cloud Learn Hub* (2020). URL: https://www.ibm.com/uk-en/cloud/learn/neural-networks.

[18] EliteAI. 'The Math Behind Decision Trees'. In: *Medium* (2021). URL: https://eliteai-coep.medium.com/the-math-behind-decision-trees-9d843b3e4057.

[19] Vlad M. Cora Eric Brochu and Nando de Freitas. 'A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning'. In: (2010). URL: https://arxiv.org/pdf/1012.2599.pdf.

[20] Sydney Firmin. 'Hyperparameter Tuning Black Magic'. In: *Alteryx* (2019). URL: https://community.alteryx.com/t5/Data-Science/Hyperparameter-Tuning-Black-Magic/ba-p/449289.

[21] Sydney Firmin. 'Hyperparameter Tuning Black Magic'. In: *Alteryx* (2019). URL: https://community.alteryx.com/t5/Data-Science/Hyperparameter-Tuning-Black-Magic/ba-p/449289.

[22] 'How to evaluate you model using the Confusion Matrix'. In: *Towards AI* (2021). URL: https://subscription.packtpub.com/book/data/9781838555078/6/ch06lvl1sec34/confusion-matrix.

[23] Guolin Ke. 'LightGBM'. In: *Wikipedia* (2022). URL: https://en.wikipedia.org/wiki/LightGBM.

[24] Takuya Akiba; Shotaro Sano; Toshihiko Yanase; Takeru Ohta; Masanori Koyama. 'Optimize Your Optimization'. In: *Optuna* (2019). URL: https://optuna.org/.

[25] H. J. Kushner. 'A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise'. In: (1964).

[26] 'Logistic regression'. In: *Wikipedia* (2022). URL: https://en.wikipedia.org/wiki/Logistic_regression.

[27] Ask Moe Løite. 'Binary Classification of Credit Card Users with Logistic Regression, XGBoost and LightGBM'. In: *Project Thesis NTNU* (2022).

[28] Leon Lok. 'Decision Trees, Random Forests and Gradient Boosting: What's the Difference?' In: (2022). URL: https://leonlok.co.uk/blog/decision-trees-random-forests-gradient-boosting-whats-the-difference/.

[29] Olivier Bousquet ; Stephane Boucheron ; Gabor Lugosi. 'Introduction to Statistical Learning Theory'. In: (). URL: http://www.econ.upf.edu/~lugosi/mlss_slt.pdf.

[30] Scott M. Lundberg. 'Topical Overviews'. In: (2018). URL: https://shap.readthedocs.io/en/latest/overviews.html.

[31] Ismail Mebsout. 'Deep Learning's mathematics'. In: *Towards Data Science* (2020). URL: https://towardsdatascience.com/deep-learnings-mathematics-f52b3c4d2576.

[32] Gustavo A. Lujan-Moreno; Phillip R. Howard; Omar G. Rojas; Douglas C. Montgomery. 'Topical Overviews'. In: (2018). URL: https://www.sciencedirect.com/science/article/abs/pii/S0957417418303178.

[33] Aratrika Pal. 'Gradient Boosting Trees for Classification: A Beginner's Guide'. In: *The Startup* (2020). URL: https://medium.com/swlh/gradient-boosting-trees-for-classification-a-beginners-guide-596b594a14ea.

[34] Terence Parr and Jeremy Howard. 'How to explain gradient boosting'. In: *explained.ai* (). URL: https://explained.ai/gradient-boosting/.

[35] Bakkeli Nan Zou Poppe Christian Borgeraas Elling. 'Lånefinansiert forbruk i Norge anno 2019'. In: (2019). URL: https://oda.oslomet.no/oda-xmlui/handle/20.500.12199/2981.

[36] Rukshan Pramoditha. 'The Concept of Artificial Neurons (Perceptrons) in Neural Networks'. In: *Towards Data Science* (2021). URL: https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc.

[37] Scikit-learn. 'Logistic regression'. In: *Scikit-learn* (2023). URL: https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression.

[38] Su-In Lee Scott M. Lundberg. 'A Unified Approach to Interpreting Model Predictions'. In: (2017). URL: https://arxiv.org/pdf/1705.07874.pdf.

[39] Riccardo Di Sipio. 'A Quick Guide to AUC-ROC in Machine Learning Models'. In: *Medium* (2021). URL: https://towardsdatascience.com/a-quick-guide-to-auc-roc-in-machine-learning-models-f0aedb78fbad.

[40] Marguerite Ennis ; Geoffrey Hinton ; David Naylor ; Mike Revow ; Robert Tibshirani. 'A comparison of statistical learning methods on the GUSTO database'. In: *Statistics in Medicine* (1998). URL: https://onlinelibrary.wiley.com/doi/10.1002/(SICI)1097-0258(19981115)17:21%3C2501::AID-SIM938%3E3.0.CO;2-M.

[41] Amy Tikkanen. 'credit card'. In: *Britannica* (2022). URL: https://www.britannica.com/topic/credit-card.

[42] Jerome Friedman Trevor Hastie Robert Tibshirani. 'The Elements of Statistical Learning Data Mining, Inference, and Prediction, Second Edition'. In: (2009). URL: https://link.springer.com/book/10.1007/978-0-387-84858-7.

[43] Wikipedia. 'scikit-learn'. In: *Wikipedia* (2023). URL: https://en.wikipedia.org/wiki/Scikit-learn.

[44] David Chen ; Echo Yang. 'Tree-Math'. In: *Github* (2020). URL: https://github.com/YC-Coder-Chen/Tree-Math/blob/master/LightGBM.md.

Appendix A

Variables in the different datasets with explanation

A.1 The fundamental dataset

Table A.1: Explanation of all the variables in the fundamental dataset

Variable Name	Observations missing	Explanation
Utgangspunkt	0%	The predicting month
BK_ACCOUNT_ID	0%	Internal account ID
ProductId	0%	What kind of credit card the customer got
Revolver	0%	An individual who carries a balance from month to month
Fullpayer	0%	An individual who pays in each time to avoid interests
FørsteBrukt	0%	First time the credit card was used
Førstekortbruk	0%	Also first time the credit card was used
SisteKortbruk	0%	Last time the credit card is used
SisteTransaksjon	0%	Last transaction done
AktivEtterPassiv	0%	The response
Alder	0%	The age of the customer
Kjønn	0%	The sex of the customer

A.2 The appliance dataset

Table A.2: Explanation of all the variables in the appliance dataset

Variable Name	Observations missing	Explanation
BK_ACCOUNT_ID	0	Internal account ID
CREATED_DT	0%	The date, and the time of the day the appliance was filed
ApplicationSalesChannel	0%	At what service the customer applied for the card
APPLIED_CREDIT _LIMIT AMT	0%	The balance the customer applied for
GRANTED_CREDIT _LIMIT AMT	0%	The balance the customer got

GROSS_INCOME_AMT	0%	The gross income for the customer
STUDENT_LOAN_AMT	0%	The amount of student-loan the customer has
MORTGAGES_AMT	0%	The amount of house-loan the customer has
EMPLOYMENT_TYPE _NAME	0%	A categorical feature that tells the type of employment of each customer
EMPLOYMENT _DURATION_DESC	0%	A categorical feature that tells how long the customer has been employed at that employment
HABITATION_TYPE _NAME	0%	A categorical feature that tells how the customer lives
MARITAL_STATUS _NAME	0%	A categorical feature that tells the marital status of the customer
DebtRegisterNum	41%	Is the number of credit cards the customer has
DebtRegisterIELA	41%	The credit card debt assosiated with each customer
TAX_CLASS_CD	6%	The tax class the customer belonged to last year
LastTaxYear2_TAX _CLASS_CD	12%	The tax class the customer belonged to 2 years ago
LastTaxYear3_TAX _CLASS_CD	17%	The tax class the customer belonged to 3 years ago
HOMEOWNER_IND	0%	Describes if the customer owns a home
HOUSING _COOPERATIVE_IND	0%	An indicator if the customer is part of a housing co-operative
NoOfChildren	0%	The number of children each customer has
FLI_AMT	0%	A liquidity indicator made by Sparebank1 on how well the customer can handle a loan
SFLI_AMT	0%	A stress test on FLI AMT where the intrests are raised
SumAvailable	9%	The amount of money available to the customer

A.3 The historical credit card dataset

Table A.3: Explanation of all the variables in the historical credit card dataset

Variable Name	**Observations missing**	**Explanation**
BK_ACCOUNT_ID	0	Internal account ID
PeriodId	0%	The period this information is relevant
AIRLINE_12	0%	How much money the customer has spent on airlines in the last 12 months with their credit card
ELECTRIC _APPLIANCE_12	0%	How much money the customer has spent on electric appliances in the last 12 months with their credit card
FOOD_STORES _WAREHOUSE_12	0%	How much money the customer has spent on groceries in the last 12 months with their credit card
HOTEL_MOTEL_12	0%	How much money the customer has spent on hotels and motels in the last 12 months with their credit card
HARDWARE_12	0%	How much money the customer has spent on hardware in the last 12 months with their credit card
INTERIOR _FURNISHINGS_12	0%	How much money the customer has spent on interior and furnishing in the last 12 months with their credit card
OTHER_RETAIL_12	0%	How much money the customer has spent on airlines in the last 12 months with their credit card

OTHER_SERVICES_12	0%	How much money the customer has spent on other services that is not included in the other variables in the last 12 months with their credit card
OTHER _TRANSPORT_12	0%	How much money the customer has spent on transportation in the last 12 months with their credit card
RECREATION_12	0%	How much money the customer has spent on recreation in the last 12 months with their credit card
RESTAURANTS _BARS_12	0%	How much money the customer has spent on restaurants and bars in the last 12 months with their credit card
SPORTING_TOY _STORES_12	0%	How much money the customer has spent on sporting and toy stores in the last 12 months with their credit card
TRAVEL_AGENCIES_12	0%	How much money the customer has spent on travel agencies in the last 12 months with their credit card
VEHICLES_12	0%	How much money the customer has spent on vehicles in the last 12 months with their credit card
QUASI_CASH_12	0%	How much money the customer has spent on quasi cash transactions in the last 12 months with their credit card
AIRLINE_3	0%	How much money the customer has spent on airlines in the last 3 months with their credit card
ELECTRIC _APPLIANCE_12	0%	How much money the customer has spent on electric appliances in the last 12 months with their credit card
FOOD_STORES _WAREHOUSE_3	0%	How much money the customer has spent on groceries in the last 3 months with their credit card
HOTEL_MOTEL_3	0%	How much money the customer has spent on hotels and motels in the last 3 months with their credit card
HARDWARE_3	0%	How much money the customer has spent on hardware in the last 3 months with their credit card
INTERIOR _FURNISHINGS_3	0%	How much money the customer has spent on interior and furnishing in the last 3 months with their credit card

A.4 The historical transactions dataset

Table A.4: Explanation of all the variables in the historical transactions dataset

Variable Name	**Observations missing**	**Explanation**
BK_ACCOUNT_ID	0	Internal account ID
PeriodId	0%	The period this information is relevant
SumPaidToCCL12	0%	Sum paid from the customers bank account to known credit card accounts the last 12 months
SumPaidToRepayment LoanL12	0%	Sum paid from the customers bank account to repayment of loan the last 12 months

CountPaidTo Repayment LoanL12	69%	Number of payments from the customers bank account to repayment of loan the last 12 months
CountPaidToCCL12	31%	Number of payments from the customers bank account to known credit card accounts the last 12 months
CountDistinct PaidToRepayment LoanL12	69%	Number of payments from the customers bank account to repayment of distinct external loans the last 12 months
CountDistinct PaidToCCL12	31%	Number of payments from the customers bank account to known distinct external credit card accounts the last 12 months
CountRoundPaid ToRepaymentLoanL12	46%	Number of round payments, (dividable by 100), from the customers bank account to repayment of loan the last 12 months
CountRound PaidToCCL12	9%	Number of round payments, (dividable by 100), from the customers bank account to known credit card accounts the last 12 months

Appendix B

Correlation plot

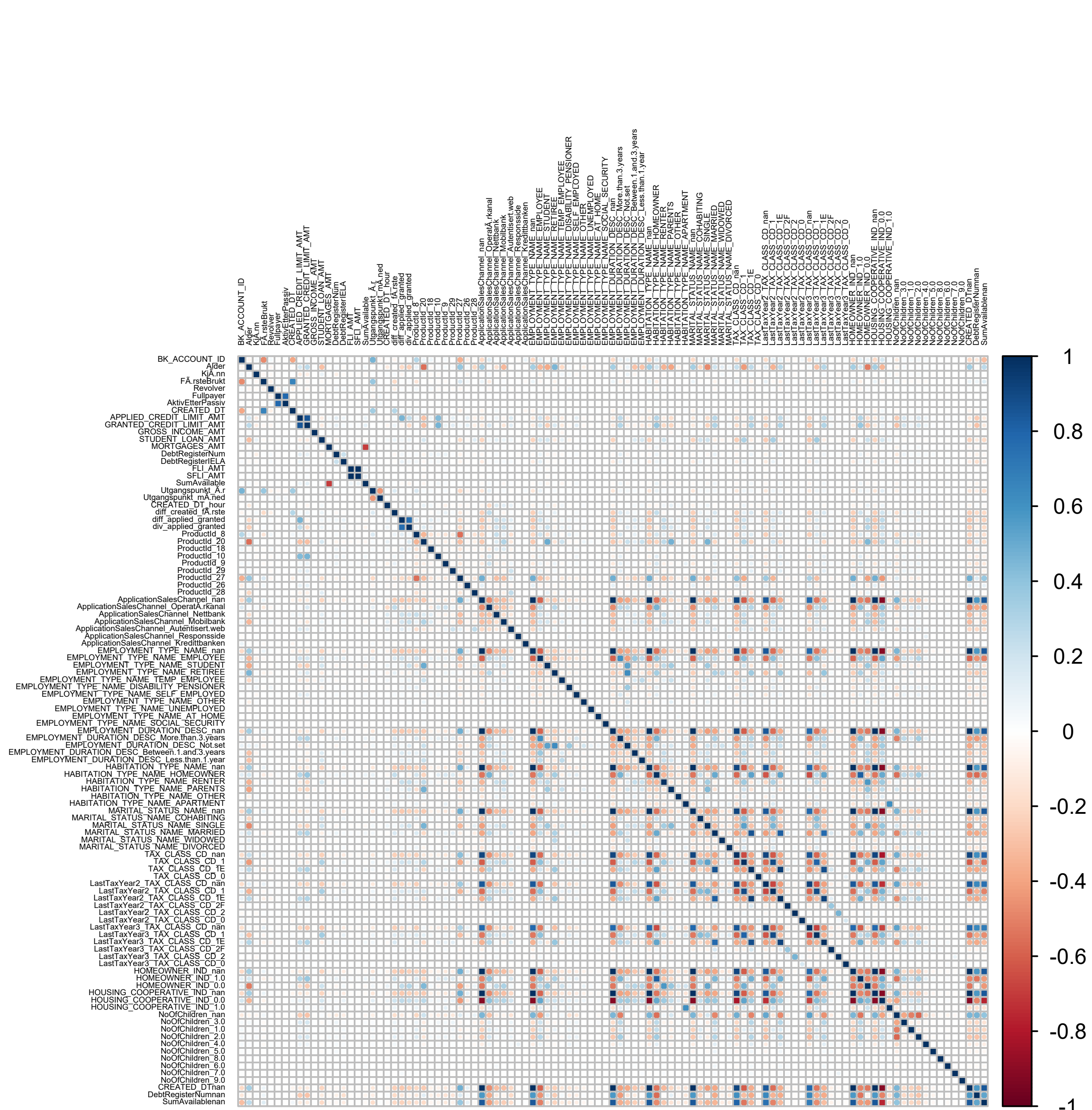

Figure B.1: Correlations in the merged dataset between the fundamental dataset and the appliance dataset after pre-processing

Appendix C

Results Logistic Regression

C.1 Code Printout

```
The execution time is: 38159.651361227036
        Best value (Score): 0.56965
        Best params:
                penalty: l2
                C: 0.25047768427161093
                tol: 3.8726594017247394e-08
                solver: saga
                max_iter: 35
```

Figure C.1: Printout of the best hyperparameter values, the best objective value and the execution time(s), for the first optimization with Logistic Regression

```
The execution time is: 10729.942382335663
        Best value (Score): 0.56965
        Best params:
                C: 0.2333449360609842
                tol: 0.0001311102130259688
                max_iter: 54
```

Figure C.2: Printout of the best hyperparameter values, the best objective value and the execution time(s), for the second optimization with Logistic Regression

C.2 Optimization Plots First Optimization

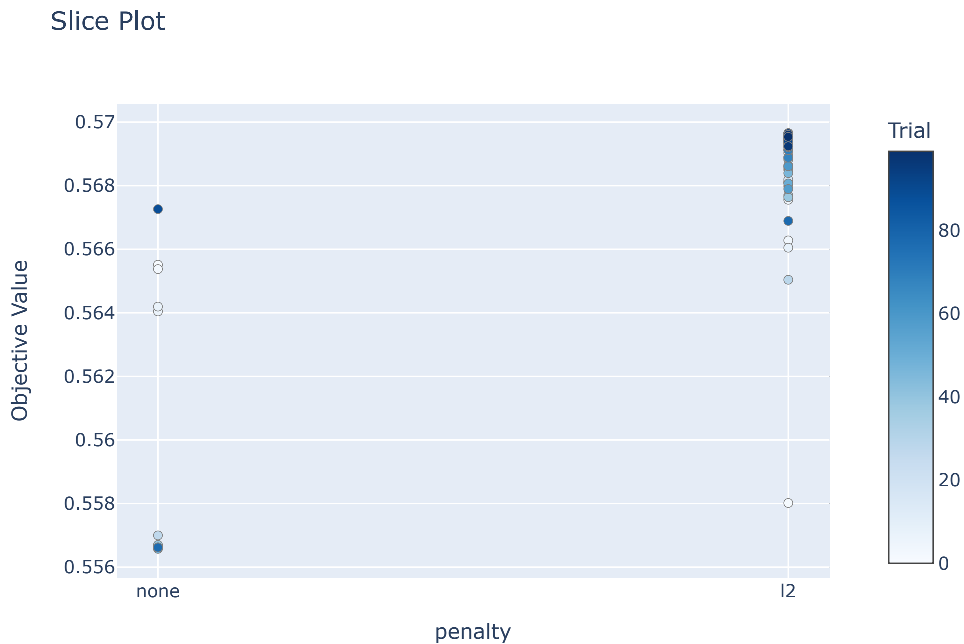

Figure C.3: Slice plot showing the impact the hyperparameter "penalty" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

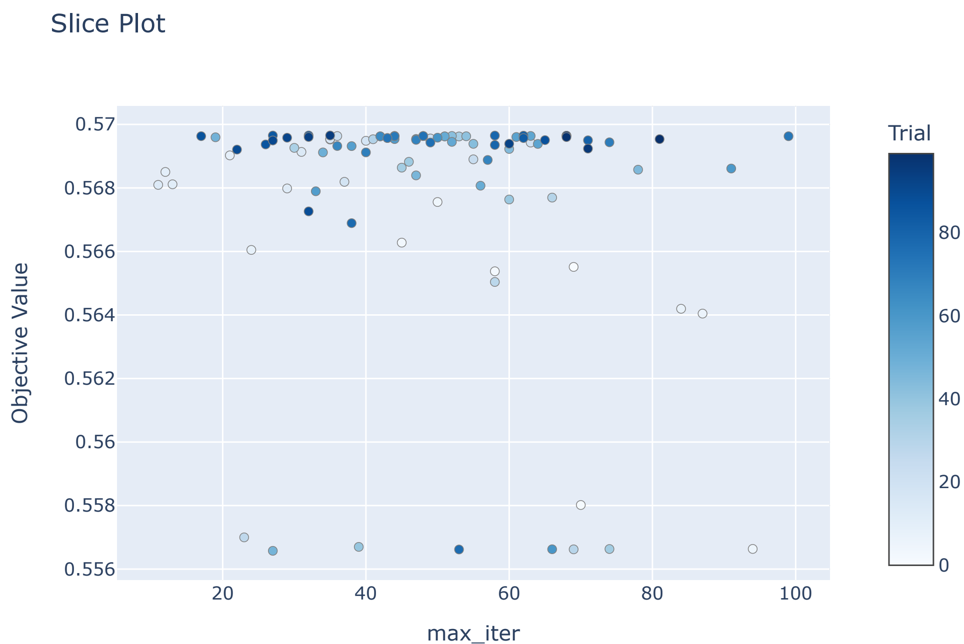

Figure C.4: Slice plot showing the impact the hyperparameter "max_iter" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

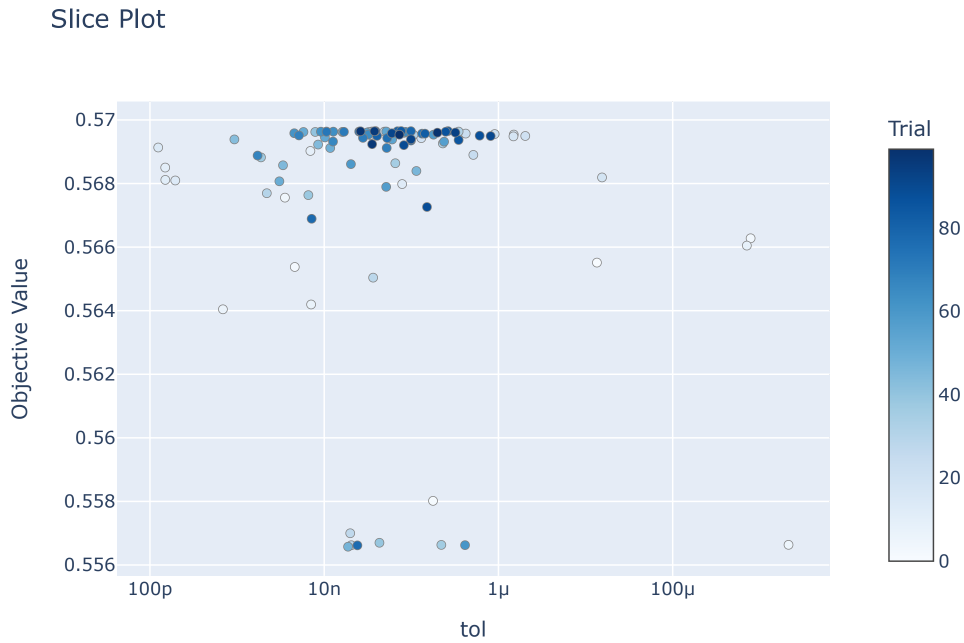

Figure C.5: Slice plot showing the impact the hyperparameter ”tol” has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

C.3 Optimization Plots Second Optimization

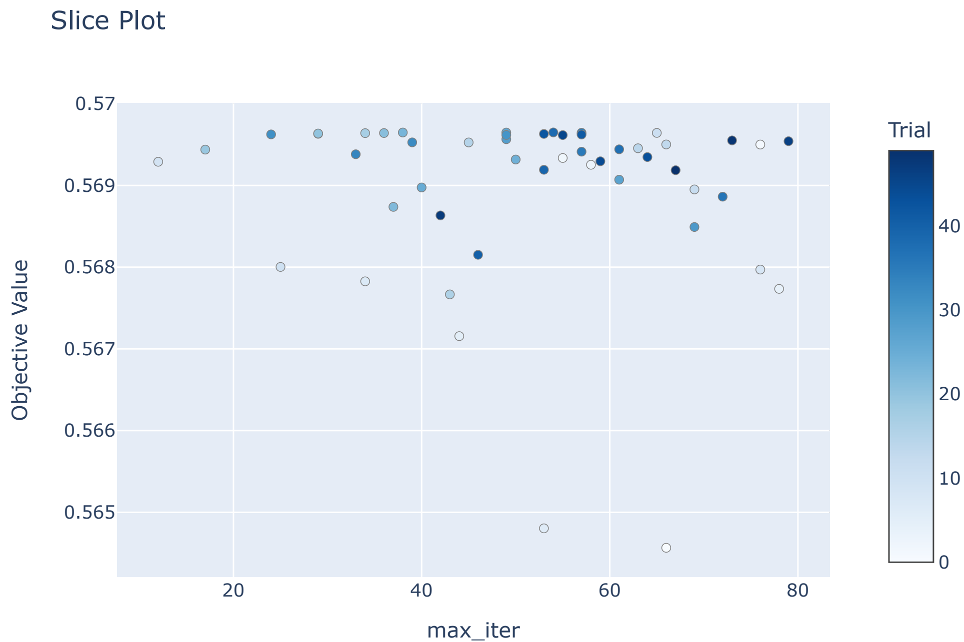

Figure C.6: Slice plot showing the impact the hyperparameter ”max_iter” has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

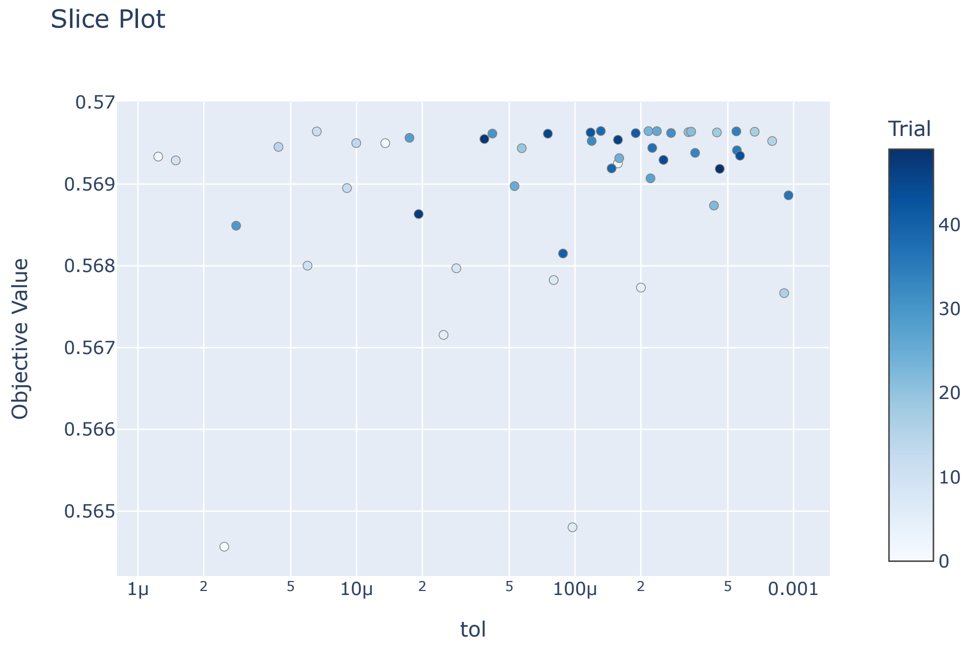

Figure C.7: Slice plot showing the impact the hyperparameter "tol" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

Appendix D

Results Gradient Boosted Decision Trees

D.1 Code Printout

```
The execution time is: 30687.894843816757
        Best value (Score): 0.65689
        Best params:
                n_estimators: 2272
                learning_rate: 0.0010180592137837522
                num_leaves: 759
                max_depth: 20
                min_child_samples: 57
                reg_alpha: 0.0037022924110828533
                reg_lambda: 0.7099353086700094
                min_split_gain: 0.9187319657459281
                subsample: 0.5025515010962769
                colsample_bytree: 0.23028910864717372
```

Figure D.1: Printout of the best hyperparameter values, the best objective value and the execution time(s), for the first optimization with Gradient Boosted Decision Trees

```
The execution time is: 41306.86256670952
        Best value (Score): 0.65916
        Best params:
                n_estimators: 2789
                learning_rate: 0.0005807394611462965
                max_depth: 30
```

Figure D.2: Printout of the best hyperparameter values, the best objective value and the execution time(s), for the second optimization with Gradient Boosted Decision Trees

D.2 Optimization Plots First Optimization

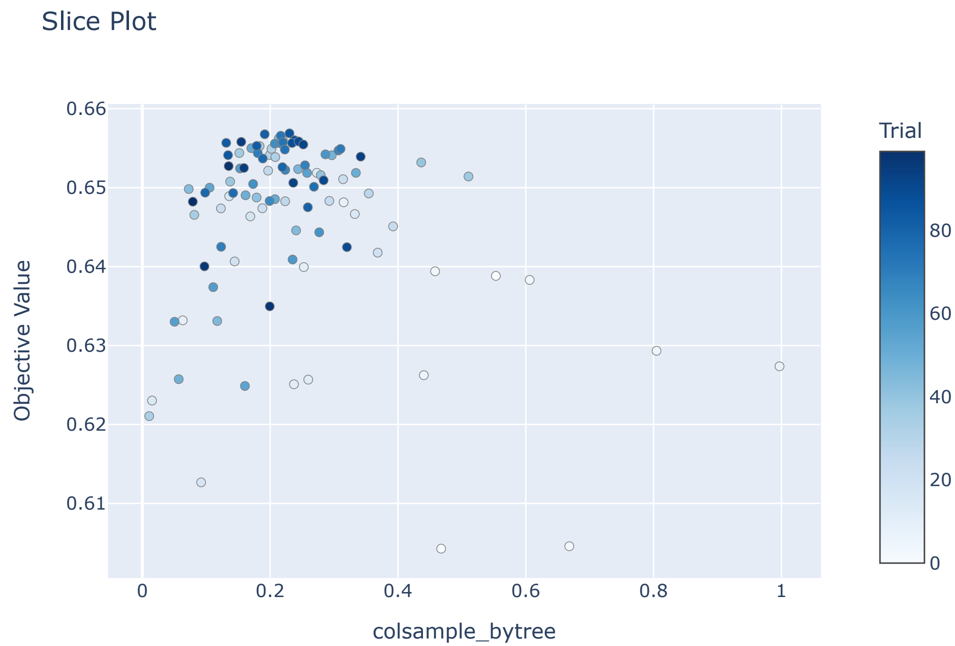

Figure D.3: Slice plot showing the impact the hyperparameter "colsample_bytree" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

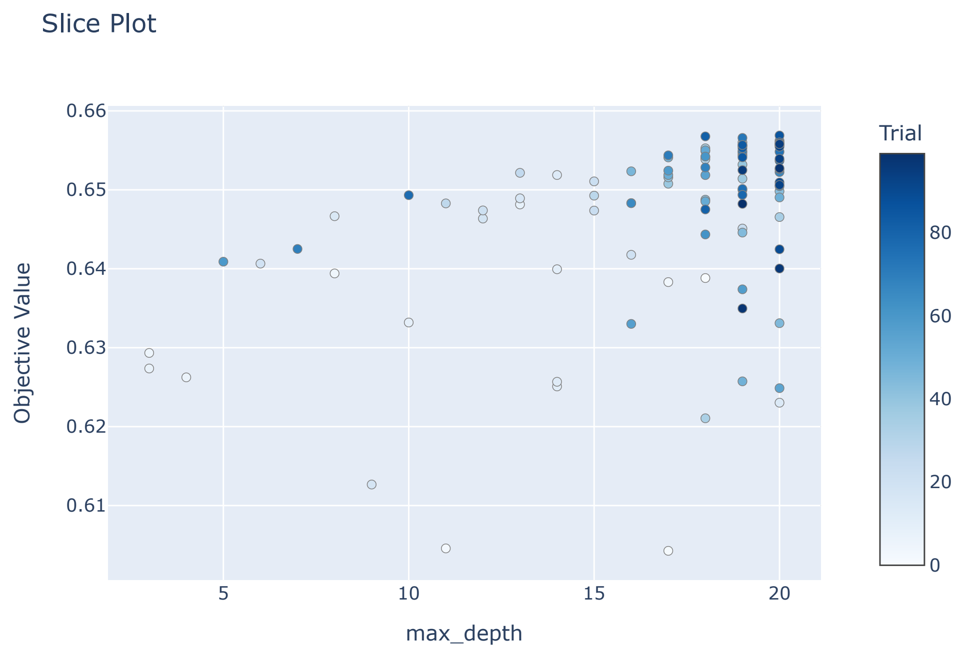

Figure D.4: Slice plot showing the impact the hyperparameter "max_depth" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

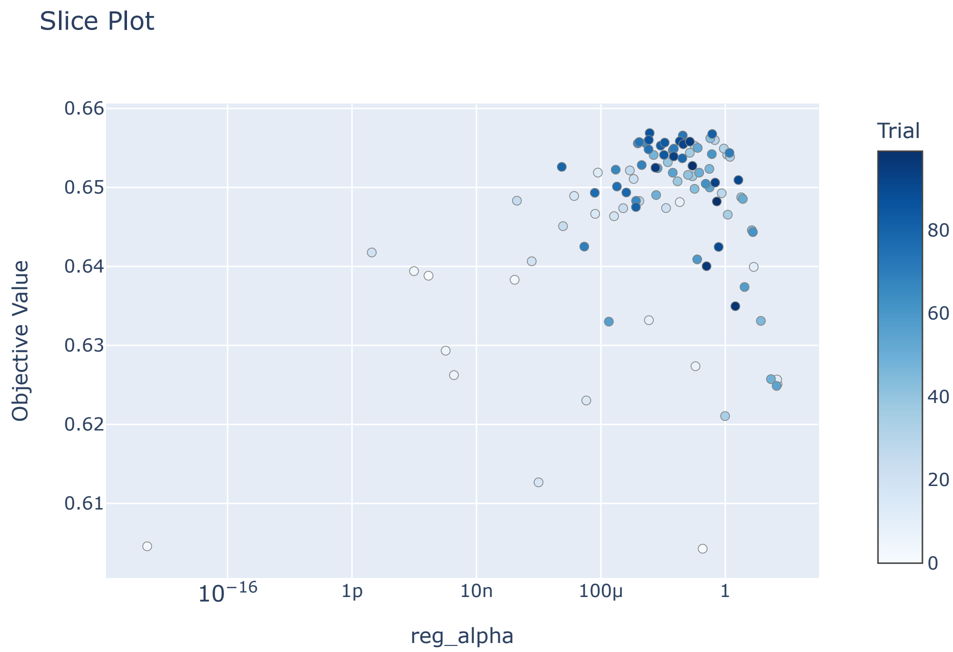

Figure D.5: Slice plot showing the impact the hyperparameter "reg_alpha" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

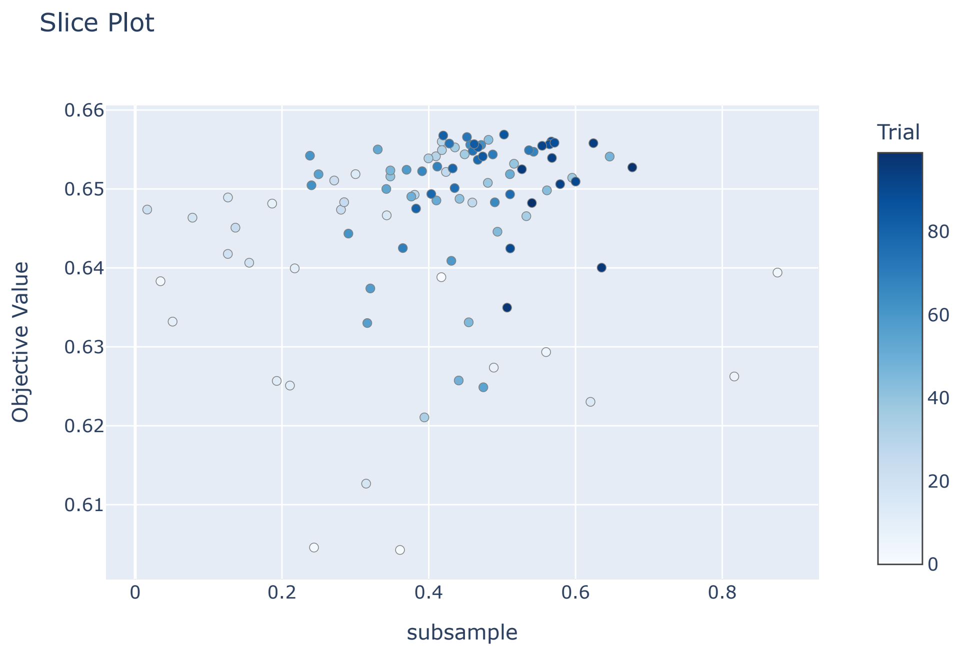

Figure D.6: Slice plot showing the impact the hyperparameter "subsample" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

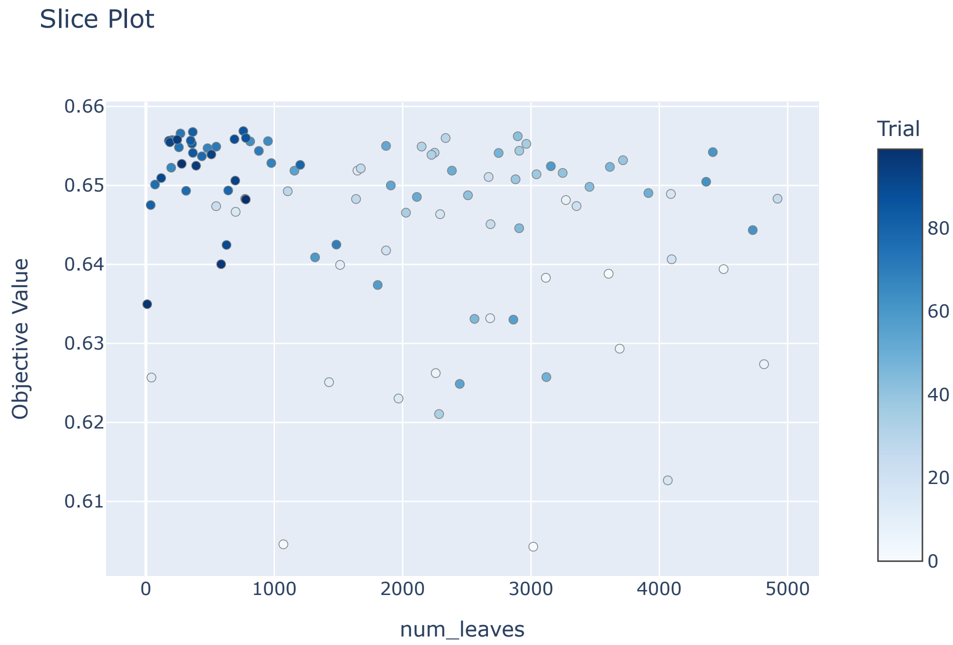

Figure D.7: Slice plot showing the impact the hyperparameter ”num_leaves” has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

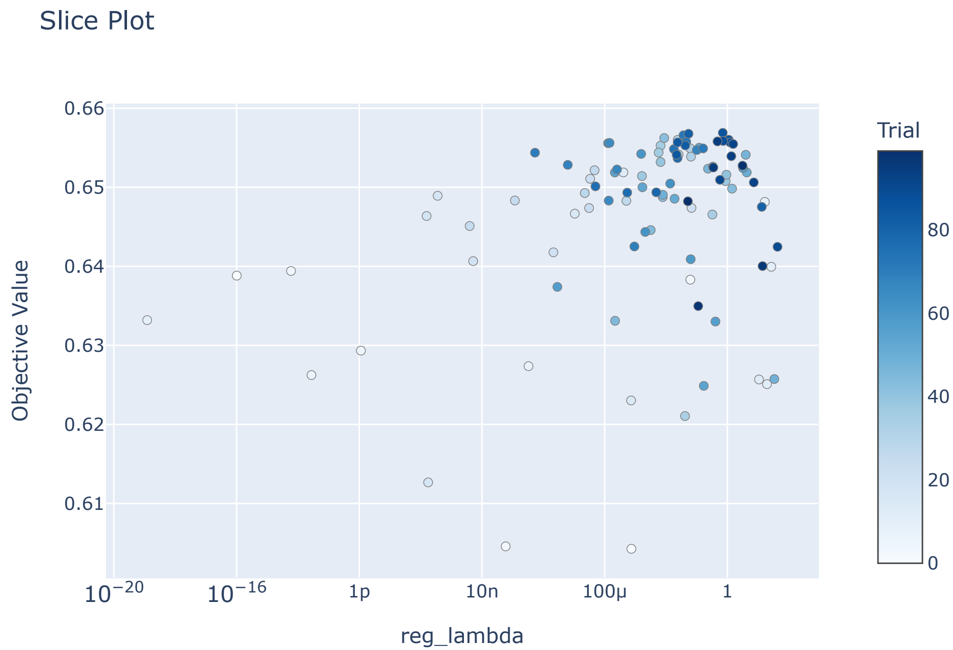

Figure D.8: Slice plot showing the impact the hyperparameter ”reg_lambda” has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

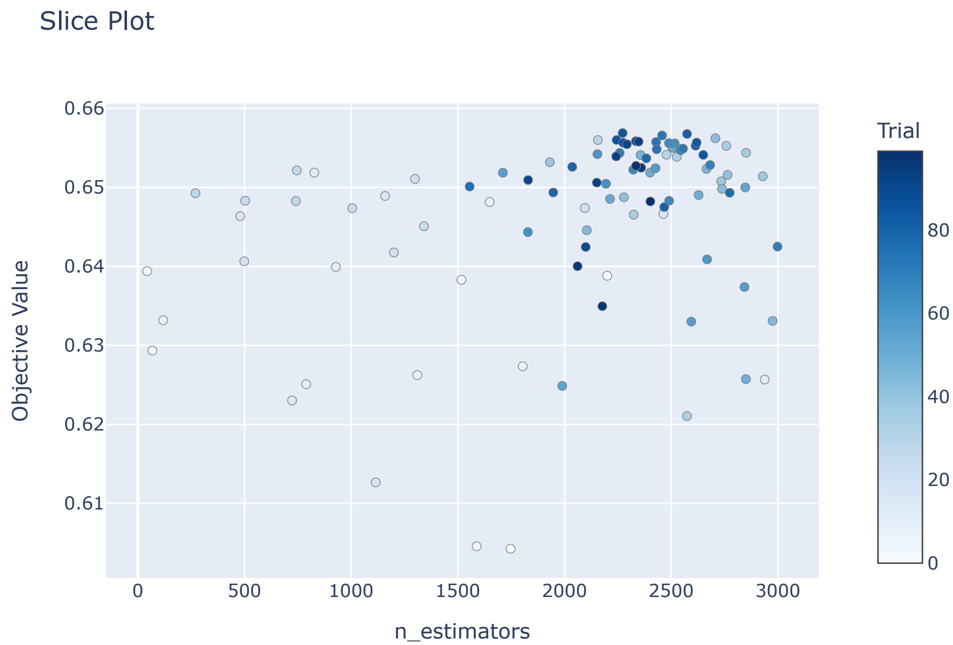

Figure D.9: Slice plot showing the impact the hyperparameter "n_estimators" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

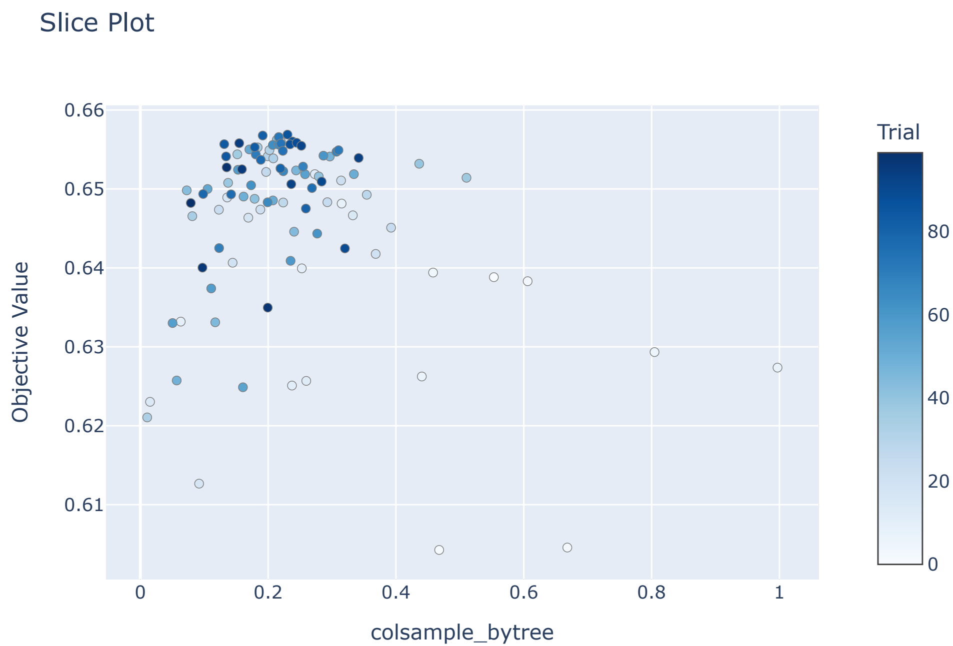

Figure D.10: Slice plot showing the impact the hyperparameter "colsample_bytree" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

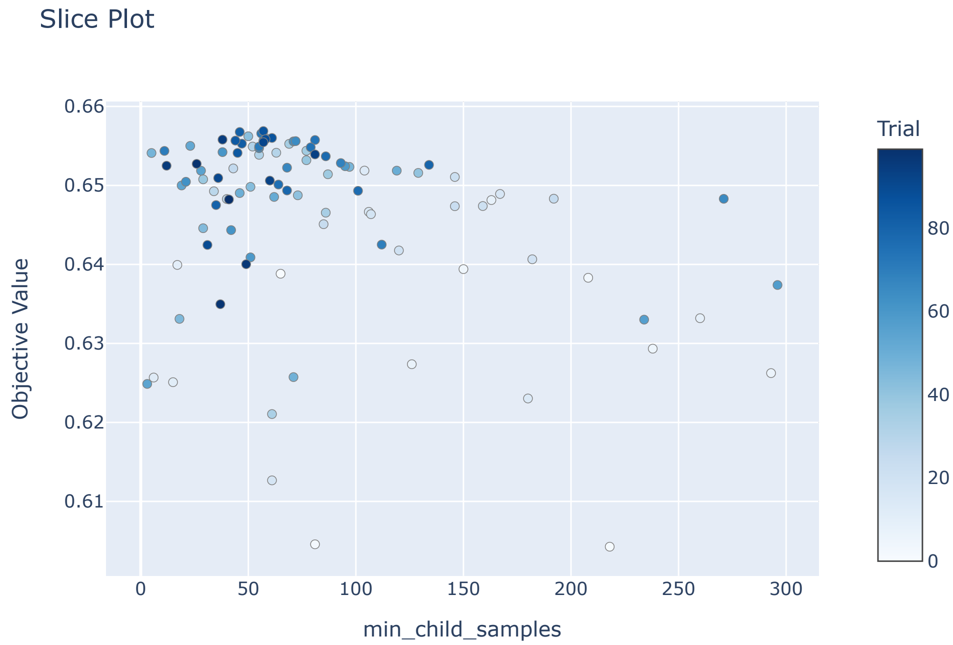

Figure D.11: Slice plot showing the impact the hyperparameter "min_child_samples" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

D.3 Optimization Plots Second Optimization

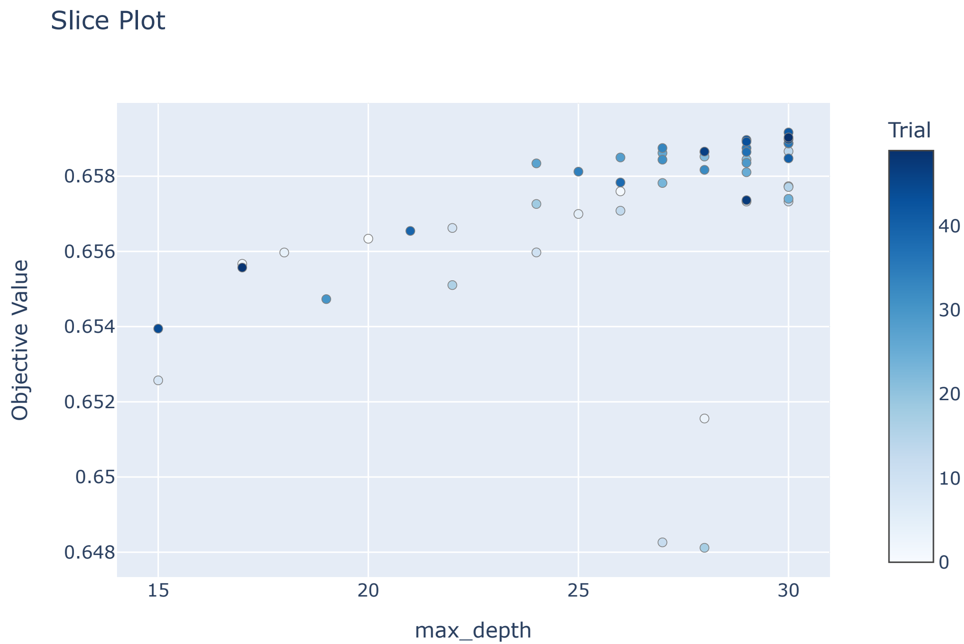

Figure D.12: Slice plot showing the impact the hyperparameter "max_depth" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

Figure D.13: Slice plot showing the impact the hyperparameter "n_estimators" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

Appendix E

Results Deep Learning

E.1 Code Printout

```
The execution time is: 191177.01587724686
        Best value (Score): 0.59169
        Best params:
                num_layers: 2
                neurons_per_layer: 169
                activation: logistic
                alpha: 0.0009155620098305521
                learning_rate_init: 0.0023164263237247782
                tol: 1.3651202992461953e-06
                max_iter: 62
```

Figure E.1: Printout of the best hyperparameter values, the best objective value and the execution time(s), for the first optimization with Deep Learning

```
The execution time is: 166233.3895394802
        Best value (Score): 0.59227
        Best params:
                num_layers: 2
                neurons_per_layer: 274
                max_iter: 85
```

Figure E.2: Printout of the best hyperparameter values, the best objective value and the execution time(s), for the second optimization with Deep Learning

E.2 Optimization Plots First Optimization

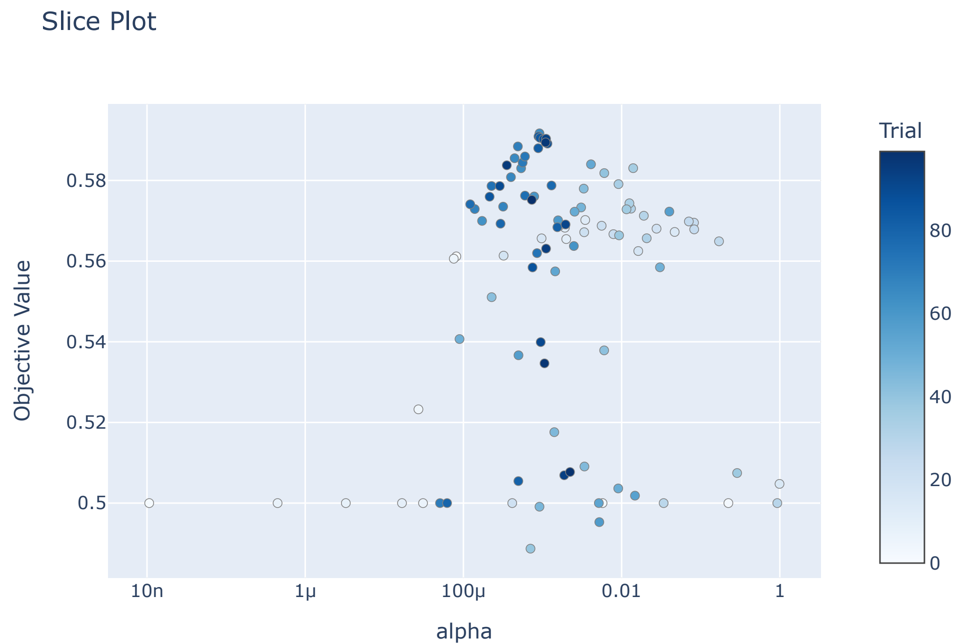

Figure E.3: Slice plot showing the impact the hyperparameter "alpha" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

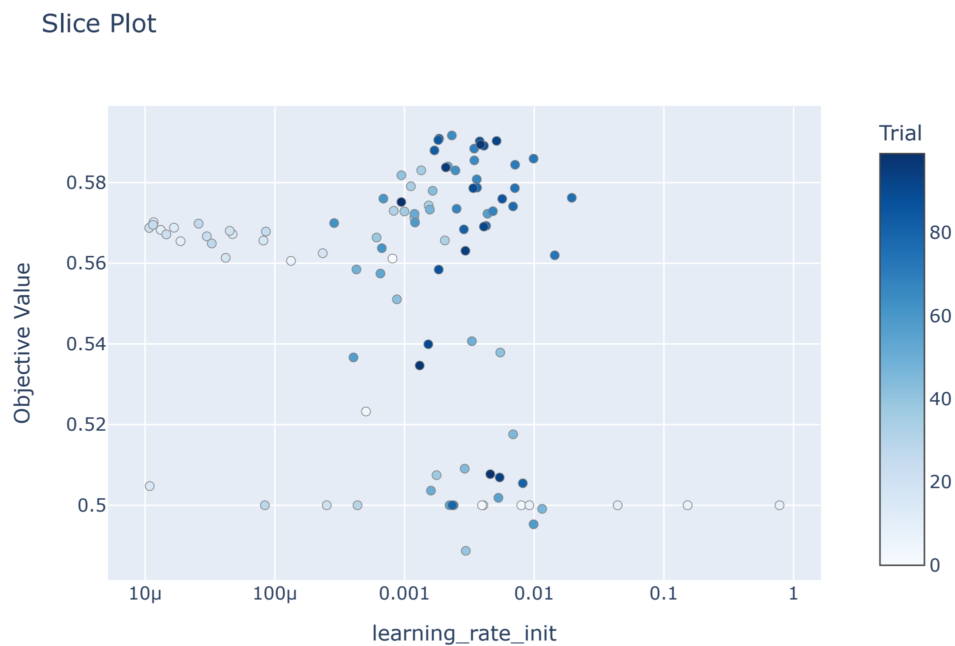

Figure E.4: Slice plot showing the impact the hyperparameter "learning_rate_init" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

Figure E.5: Slice plot showing the impact the hyperparameter "max_iter" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

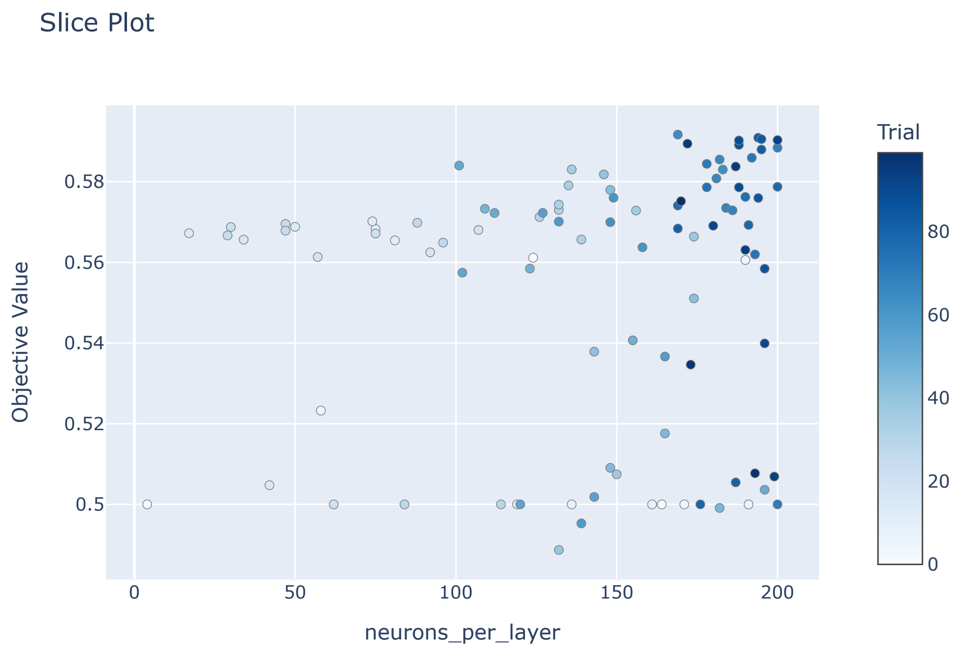

Figure E.6: Slice plot showing the impact the hyperparameter "neurons_per_layer" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

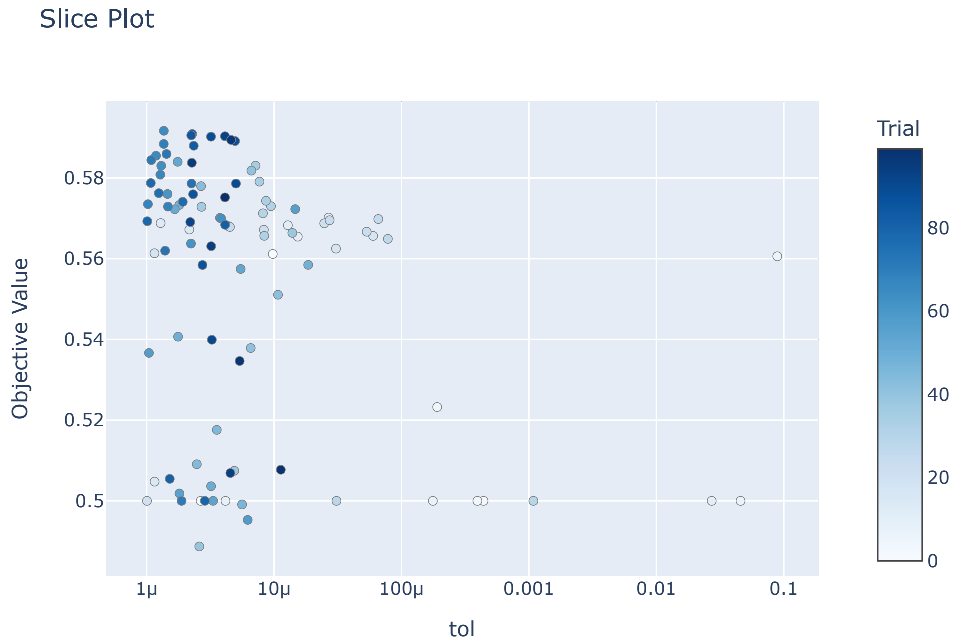

Figure E.7: Slice plot showing the impact the hyperparameter "tol" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

E.3 Optimization Plots Second Optimization

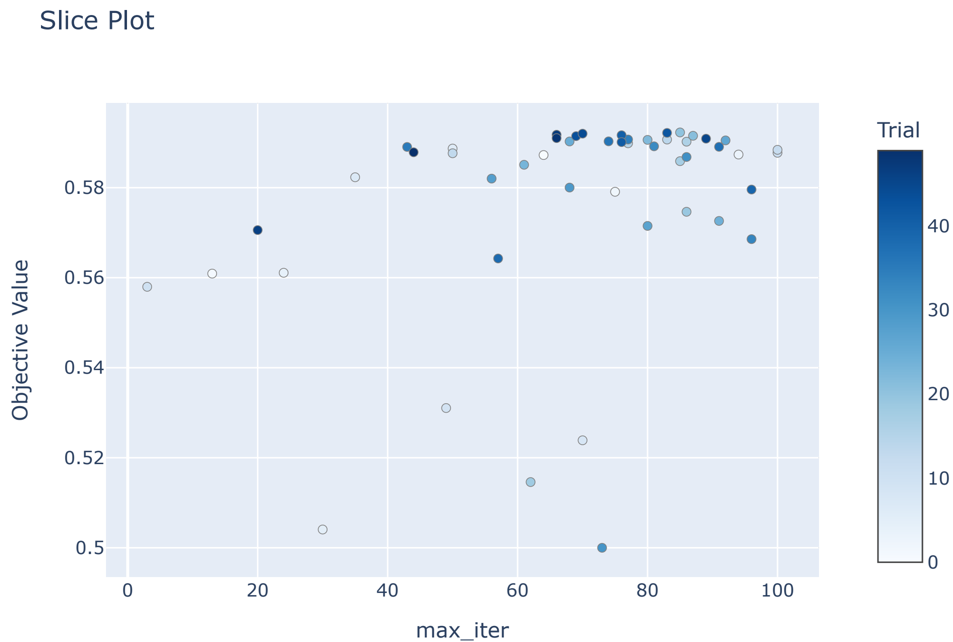

Figure E.8: Slice plot showing the impact the hyperparameter "max_iter" has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

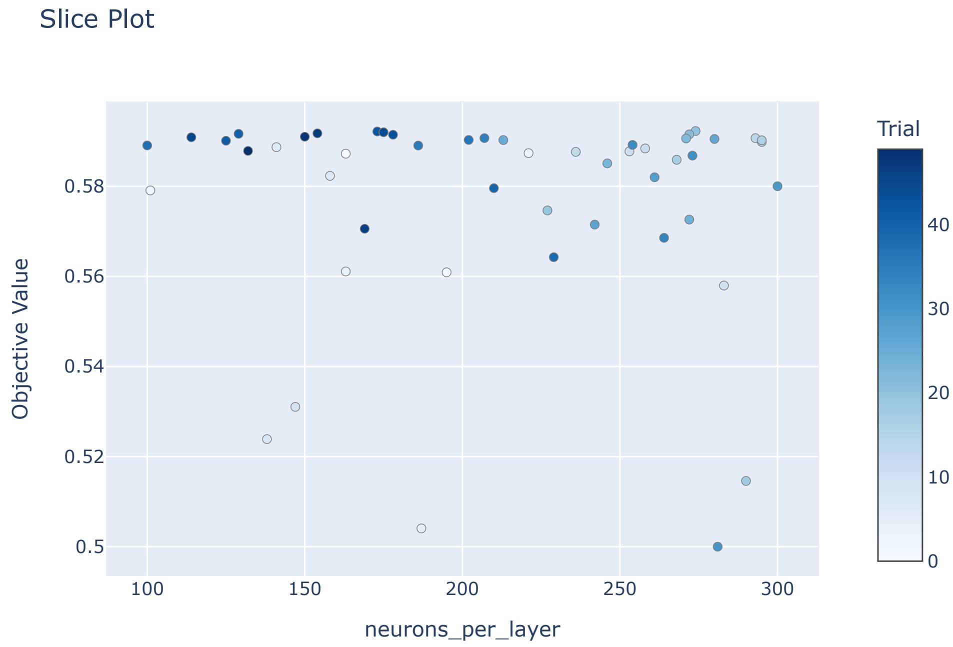

Figure E.9: Slice plot showing the impact the hyperparameter ”neurons_per_layer” has on the objective value. Whiter dots represents earlier iterations in the Bayesian optimization.

Appendix F

Other outprints and plots

F.1 Results

```
AUC:
Gradient Boosting: 0.6461336578253764
Logistic: 0.5730550987935368
Deep Learning: 0.5579644578687355

Brier-score:
Gradient Boosting: 0.05927266912555642
Logistic: 0.05965173362294544
Deep Learning: 0.06977085543754236

MCC:
Gradient Boosting: 0.1015857224010771
Logistic: 0.05303561602889058
Deep Learning: 0.04495052967244716

BACC:
Gradient Boosting: 0.6037646746860383
Logistic: 0.5535299072891977
Deep Learning: 0.5408271982799147

Accuracy:
Gradient Boosting: 0.5411453100739794
Logistic: 0.5817274028069535
Deep Learning: 0.709821292659908

Specificity:
Gradient Boosting: 0.5319695589957071
Logistic: 0.5858592428775855
Deep Learning: 0.7345843632106154

Sensitivity:
Gradient Boosting: 0.6755597903763697
Logistic: 0.5212005717008099
Deep Learning: 0.34707003334921394
```

```
AUC:
Gradient Boosting: 0.660573633020934
Logistic: 0.5727732311867366
Deep Learning: 0.5966923589556451

Brier-score:
Gradient Boosting: 0.05932061308443945
Logistic: 0.059624957777832166
Deep Learning: 0.05943672617141246

MCC:
Gradient Boosting: 0.11904113732414362
Logistic: 0.047502346846835374
Deep Learning: 0.0671052398388796

BACC:
Gradient Boosting: 0.6216585090448952
Logistic: 0.545834453309706
Deep Learning: 0.5685825601751917

Accuracy:
Gradient Boosting: 0.5372484549578348
Logistic: 0.3670350412518647
Deep Learning: 0.5089353670045971

Specificity:
Gradient Boosting: 0.5248796669702095
Logistic: 0.3408351762716274
Deep Learning: 0.5001951346429035

Sensitivity:
Gradient Boosting: 0.7184373511195807
Logistic: 0.7508337303477847
Deep Learning: 0.6369699857074798
```

Figure F.1: Results from classification metrics on all models. To the left default hyperparameters and threshold are used. To the right tuned hyperparameters and threshold are used.

```
Matrix LGBM:
[[32714 28782]
 [ 1362  2836]]
Matrix LR:
[[36028 25468]
 [ 2010  2188]]
Matrix DL:
[[45174 16322]
 [ 2741  1457]]
```

```
Matrix LGBM:
[[32278 29218]
 [ 1182  3016]]
Matrix LR:
[[20960 40536]
 [ 1046  3152]]
Matrix DL:
[[30760 30736]
 [ 1524  2674]]
```

Figure F.2: Confusion matrices for all the models. To the left default hyperparameters and threshold are used. To the right tuned hyperparameters and threshold are used.

F.2 Threshold plots on default models

Sensitivity and Specificity with default hyperparameters:

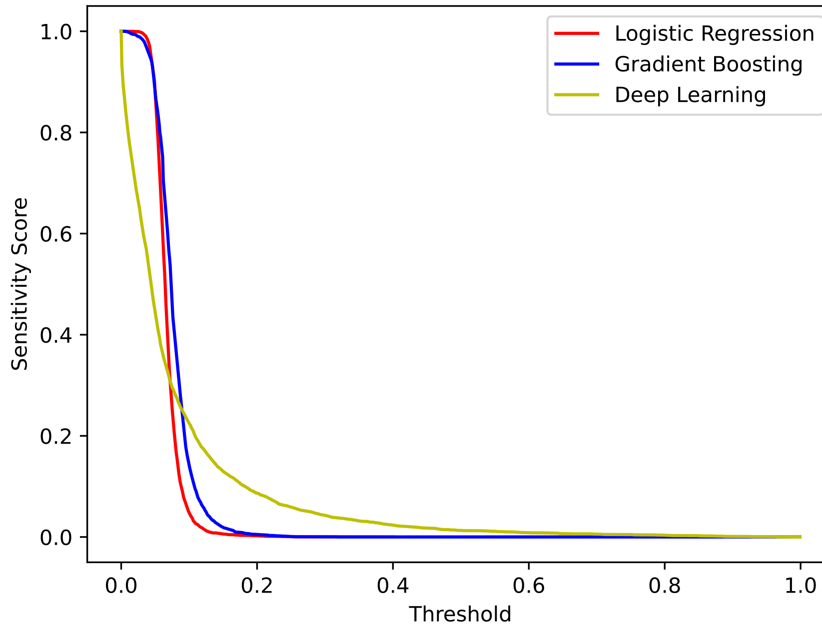

Figure F.3: A plot illustrating the impact of various threshold values on the Sensitivity for different predictive models with default hyperparameters. Logistic Regression is displayed in red, Gradient Boosted Decision Trees is displayed in blue and Deep Learning is displayed in yellow.

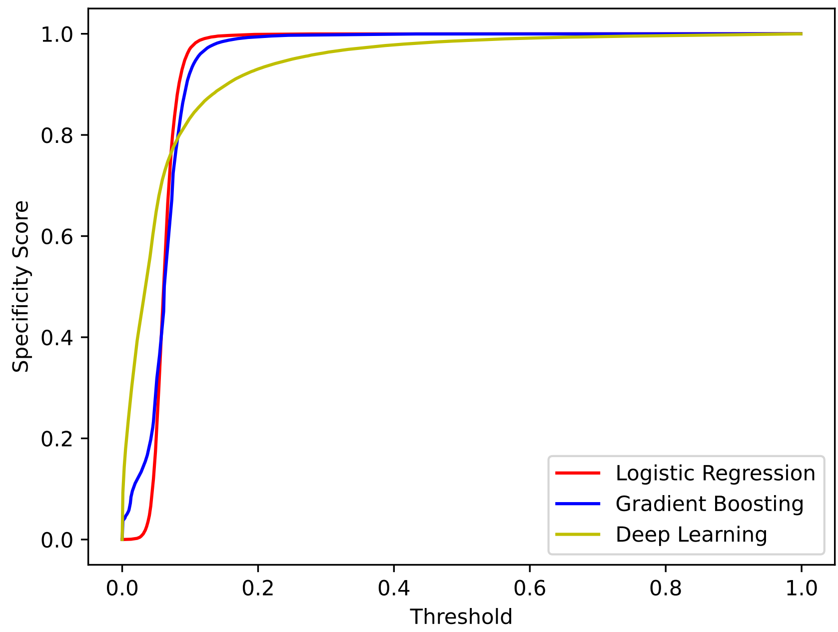

Figure F.4: A plot illustrating the impact of various threshold values on the Specificity for different predictive models with default hyperparameters. Logistic Regression is displayed in red, Gradient Boosted Decision Trees is displayed in blue and Deep Learning is displayed in yellow.

F.3 Threshold plots on tuned models

Sensitivity and Specificity with optimal hyperparameters:

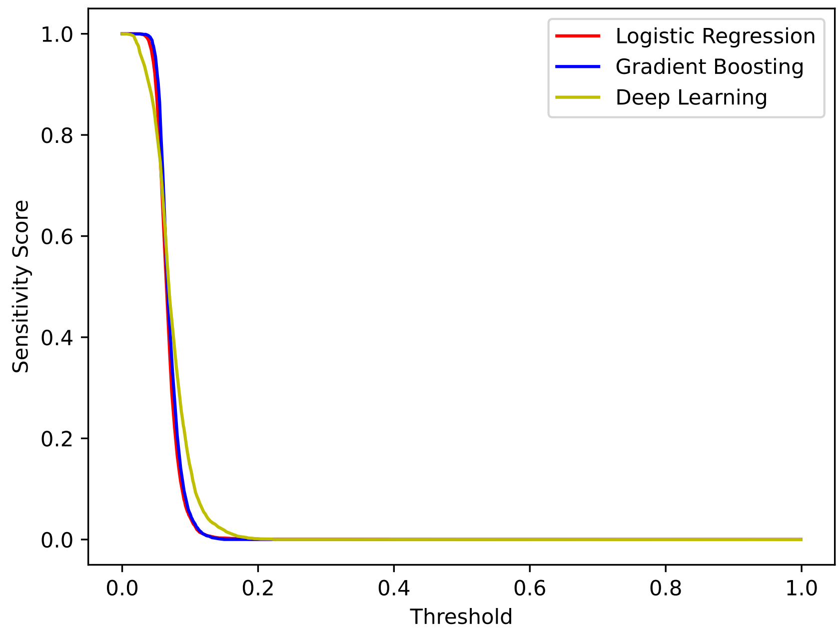

Figure F.5: A plot illustrating the impact of various threshold values on the Sensitivity for different predictive models with optimal hyperparameters. Logistic Regression is displayed in red, Gradient Boosted Decision Trees is displayed in blue and Deep Learning is displayed in yellow.

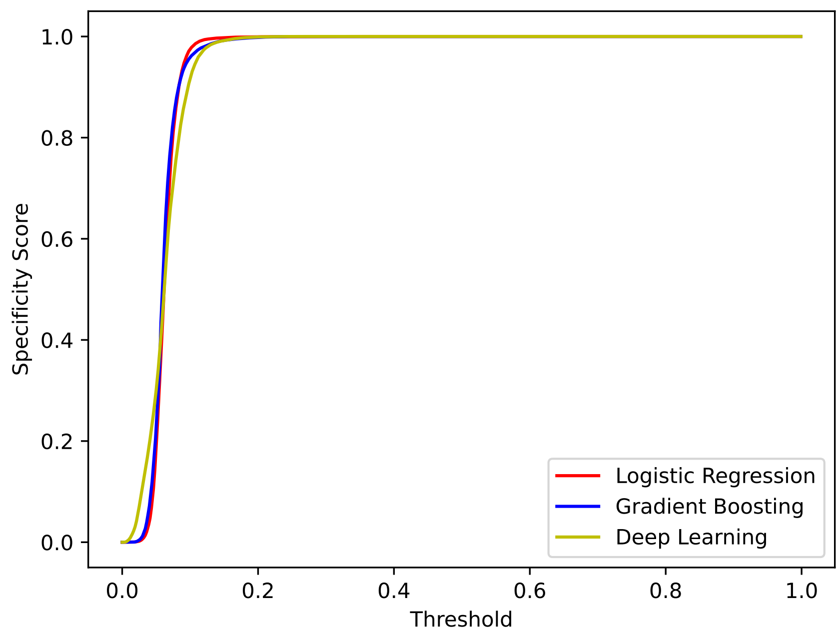

Figure F.6: A plot illustrating the impact of various threshold values on the Specificity for different predictive models with optimal hyperparameters. Logistic Regression is displayed in red, Gradient Boosted Decision Trees is displayed in blue and Deep Learning is displayed in yellow.

F.4 Printouts from the SHAP values

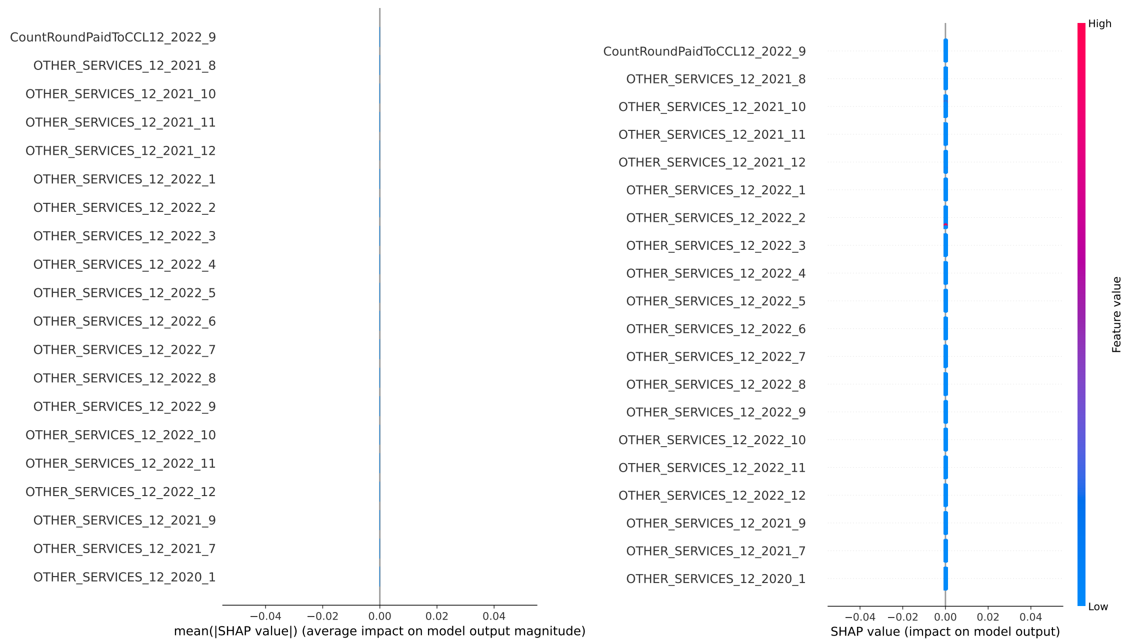

Figure F.7: A figure that should showcase the SHAP values for the tuned deep learning model.

F.4.1 Default Predicting Models

100% 65694/65694 [26:39:57<00:00, 1.47s/it]

Figure F.8: A printout of the runtime for getting the SHAP values for the default Logistic Regression model.

100% 65694/65694 [28:19:13<00:00, 1.54s/it]

Figure F.9: A printout of the runtime for getting the SHAP values for the default Gradient Boosted Decision Trees model.

100% 65694/65694 [42:33:50<00:00, 2.36s/it]

Figure F.10: A printout of the runtime for getting the SHAP values for the default Deep Learning model.

F.4.2 Tuned Predicting Models

100% 65694/65694 [26:55:49<00:00, 1.47s/it]

Figure F.11: A printout of the runtime for getting the SHAP values for the tuned Logistic Regression model.

0% 9/65694 [02:36<294:00:17, 16.11s/it]

Figure F.12: A printout of the anticipated runtime for getting the SHAP values for the tuned Gradient Boosted Decision Trees model.

100% 65694/65694 [36:51:24<00:00, 1.98s/it]

Figure F.13: A printout of the runtime for getting the SHAP values for the tuned Deep Learning model.

Appendix G

The Code

This chapter in the appendix will not contain any pre-processing. It will however contain information so that the processes that has been done with with this dataset can be repeated with another dataset.

G.1 Packages

```
from chart_studio import plotly as py
from lightgbm import LGBMClassifier
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import plotly.express as px
import optuna
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, accuracy_score
import lightgbm as lgb
from sklearn.metrics import roc_auc_score
from sklearn.metrics import brier_score_loss
from sklearn.metrics import matthews_corrcoef
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import balanced_accuracy_score
import shap
from sklearn.neural_network import MLPClassifier
```

G.2 Main Code

```
data = pd.read_csv('hele_data_fully_scaled.csv')
pd.pandas.set_option('display.max_columns',None)

data = data.drop(['Revolver'], axis=1)
data = data.drop(['Fullpayer'], axis=1)


X = data.drop(['AktivEtterPassiv'], axis=1)
```

```
y = data['AktivEtterPassiv']

X_train, X_test, y_train, y_test = train_test_split(  X, y, test_size=0.25, random_state=0)


#Training and fitting a model for Gradient Boosted Decision Trees
modelLGBM = LGBMClassifier()
modelLGBM.fit(X_train, y_train)
probsLGBM = modelLGBM.predict_proba(X_test)
probsLGBM = probsLGBM[:,1]

#Training and fitting a model for Logistic Regression
modelLR = LogisticRegression()
modelLR.fit(X_train, y_train)
probsLR = modelLR.predict_proba(X_test)
probsLR = probsLR[:,1]

#Training and fitting a model for Deep Learning
nn = MLPClassifier()
nn.fit(X_train, y_train)
probsNN = nn.predict_proba(X_test)
probsNN = probsNN[:,1]



#Making the MCC-curve
#This can be repeated with other metrics
LGBM=[]
LR=[]
KE=[]
x=[]
for i in range(1000):
    threshold=i/1000
    x.append(threshold)

    y_pred = (probsLR >= threshold).astype(int)
    LR.append(matthews_corrcoef(y_test, y_pred))



    y_pred = (probsLGBM >= threshold).astype(int)
    LGBM.append(matthews_corrcoef(y_test, y_pred))

    y_pred = (probsNN >= threshold).astype(int)
    KE.append(matthews_corrcoef(y_test, y_pred))


plt.plot(x, LR, color='r', label='Logistic Regression')
plt.plot(x, LGBM, color='b', label='Gradient Boosting')
plt.plot(x, KE, color='y', label='Deep Learning')

# Naming the x-axis, y-axis and the whole graph
plt.xlabel("Threshold")
plt.ylabel("Matthews Correlation Coefficient")
plt.legend()

plt.savefig("Threshold_Matthews_Correlation_Coefficient.pdf")
```

```
plt.show()


#Using the default threshold to classify observations
predsLGBM = (probsLGBM >= 0.064).astype(int)
predsLR = (probsLR >= 0.064).astype(int)
predsKE = (probsNN >= 0.064).astype(int)



tnLGBM, fpLGBM, fnLGBM, tpLGBM = confusion_matrix(y_test, predsLGBM).ravel()
tnLR, fpLR, fnLR, tpLR = confusion_matrix(y_test, predsLR).ravel()
tnKE, fpKE, fnKE, tpKE = confusion_matrix(y_test, predsKE).ravel()

#Prining the different scores for the different metrics
print('AUC: \nGradient Boosting:' , roc_auc_score(y_test,probsLGBM), '\nLogistic:' ,
roc_auc_score(y_test,probsLR), '\nDeep Learning:' , roc_auc_score(y_test,probsNN) )
print('\n')
print('Brier-score: \nGradient Boosting:' , brier_score_loss(y_test,probsLGBM), '\nLogistic:' ,
brier_score_loss(y_test,probsLR), '\nDeep Learning:' , brier_score_loss(y_test,probsNN) )
print('\n')

print('MCC: \nGradient Boosting:' , matthews_corrcoef(y_test,predsLGBM),  '\nLogistic:' ,
matthews_corrcoef(y_test,predsLR), '\nDeep Learning:' , matthews_corrcoef(y_test,predsKE) )
print('\n')
print('BACC: \nGradient Boosting:' , balanced_accuracy_score(y_test,predsLGBM),  '\nLogistic:' ,
balanced_accuracy_score(y_test,predsLR),
'\nDeep Learning:' , balanced_accuracy_score(y_test,predsKE) )
print('\n')
print('Accuracy: \nGradient Boosting:' , accuracy_score(y_test,predsLGBM),  '\nLogistic:' ,
accuracy_score(y_test,predsLR), '\nDeep Learning:' , accuracy_score(y_test,predsKE) )
print('\n')
print('Specificity: \nGradient Boosting:' , tnLGBM / (tnLGBM+fpLGBM),  '\nLogistic:'
, tnLR / (tnLR+fpLR), '\nDeep Learning:' , tnKE / (tnKE+fpKE) )
print('\n')
print('Sensitivity: \nGradient Boosting:' , tpLGBM / (tpLGBM+fnLGBM), '\nLogistic:'
, tpLR / (tpLR+fnLR), '\nDeep Learning:' , tpKE / (tpKE+fnKE) )
print('\n')

#Printing the confusion matrices
print('Matrix LGBM:')
print(confusion_matrix(y_true=y_test, y_pred=predsLGBM))

print('Matrix LR:')
print(confusion_matrix(y_true=y_test, y_pred=predsLR))

print('Matrix DL:')
print(confusion_matrix(y_true=y_test, y_pred=predsKE))


#Making feature importance plot for Logistig Regression
feature_importance = abs(modelLR.coef_[0])
feature_importance = 100.0 * (feature_importance / feature_importance.max())
sorted_idx = np.argsort(feature_importance)
pos = np.arange(sorted_idx.shape[0]) + .5

featfig = plt.figure()
featax = featfig.add_subplot(1, 1, 1)
```

```
featax.barh(pos[1178:], feature_importance[sorted_idx][1178:], align='center')
featax.set_yticks(pos[1178:])
featax.set_yticklabels(np.array(X.columns)[sorted_idx][1178:], fontsize=8)
featax.set_xlabel('Feature Importance')

plt.tight_layout()
plt.savefig("Feature_importance_LR.pdf")
plt.show()

#Making feature importance plot for Gradient Boosted Decision Trees

lgb.plot_importance(modelLGBM, max_num_features=20, title='')
plt.tight_layout()
plt.savefig("Feature_importance_LGBM.pdf")
plt.show()

#Making Shap plots

X_train_summary = shap.kmeans(X_train, 10)
explainer = shap.KernelExplainer(nn.predict, X_train_summary)
shap_values = explainer.shap_values(X_test)
fig = shap.summary_plot(shap_values, X_test, plot_type="bar",show=False)
plt.savefig("Feature_importance_Deep.pdf",bbox_inches="tight")
shap.summary_plot(shap_values, X_test,show=False)
plt.savefig("Feature_importance_Deep_2.pdf",bbox_inches="tight")

X_train_summary = shap.kmeans(X_train, 10)
explainer = shap.KernelExplainer(modelLGBM.predict, X_train_summary)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(shap_values, X_test,show=False)
plt.savefig("Feature_importance_shap_LGBM_2.pdf",bbox_inches="tight")
shap.summary_plot(shap_values, X_test, plot_type="bar",show=False)
plt.savefig("Feature_importance_shap_LGBM.pdf.pdf",bbox_inches="tight")


X_train_summary = shap.kmeans(X_train, 10)
explainer = shap.KernelExplainer(modelLR.predict, X_train_summary)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(shap_values, X_test,show=False)
plt.savefig("Feature_importance_shap_LR_2.pdf",bbox_inches="tight")
shap.summary_plot(shap_values, X_test, plot_type="bar",show=False)
plt.savefig("Feature_importance_shap_LR.pdf",bbox_inches="tight")
```

G.2.1 New Models

Now the same code is run again, but with these models and these thresholds:

```
param = {
    "n_estimators": 2789,
```

```
    "learning_rate": 0.0005807394611462965,
    "num_leaves": 759,
    "max_depth": 30,
    "min_child_samples": 57,
    "reg_alpha": 0.0037022924110828533,
    "reg_lambda": 0.7099353086700094,
    "min_split_gain": 0.9187319657459281,
    "subsample": 0.5025515010962769,
    "colsample_bytree": 0.23028910864717372,
    'verbose': -1,
}

modelLGBM = LGBMClassifier(**param)


param = {
    'penalty' : 'l2',
    'C'       : 0.2333449360609842,
    'tol'     : 0.0001311102130259688,
    'solver'  : 'saga',
    "max_iter": 54,
}
modelLR = LogisticRegression(**param)


num_layers = 2
neurons_per_layer = 274

param = {
    'activation' : 'logistic',
    "alpha": 0.0009155620098305521,
    "learning_rate_init": 0.0023164263237247782,
    "tol": 0.0000013651202992461953,
    "max_iter": 85,
    "hidden_layer_sizes": list(np.ones((num_layers),dtype=int)*neurons_per_layer)

}

nn = MLPClassifier(**param)



predsLGBM = (probsLGBM >= 0.06).astype(int)
predsLR = (probsLR >= 0.056).astype(int)
predsKE = (probsNN >= 0.062).astype(int)
```

G.3 Logistic Regression Tuning

This will only conain coding of the first tuning

```
def objective(trial, X_train, y_train):
```

```
    param = {
        'penalty' : trial.suggest_categorical("penalty", ["none", 'l2']),
        'C'       : trial.suggest_loguniform("C", 0.05, 100),
        'tol'     : trial.suggest_loguniform("tol", 0.0000000001, 0.01),
        'solver'  : trial.suggest_categorical("solver", ['newton-cg', 'lbfgs', 'sag','saga']),
        "max_iter": trial.suggest_int("max_iter", 10,100),
    }

    X_train, X_1, y_train, y_1 =
    train_test_split(  X_train, y_train, test_size=0.33333333333, random_state=42)
    X_2, X_3, y_2, y_3 = train_test_split(  X_train, y_train, test_size=0.5, random_state=42)

    model1 = LogisticRegression(**param)
    model2 = LogisticRegression(**param)
    model3 = LogisticRegression(**param)

    model1.fit(pd.concat([X_2,X_3]), pd.concat([y_2,y_3]))
    model2.fit(pd.concat([X_1,X_3]), pd.concat([y_1,y_3]))
    model3.fit(pd.concat([X_2,X_1]), pd.concat([y_2,y_1]))

    probs1 = model1.predict_proba(X_1)
    probs2 = model2.predict_proba(X_2)
    probs3 = model3.predict_proba(X_3)

    probs1 = probs1[:,1]
    probs2 = probs2[:,1]
    probs3 = probs3[:,1]


    auc1 = roc_auc_score(y_1,probs1)
    auc2 = roc_auc_score(y_2,probs2)
    auc3 = roc_auc_score(y_3,probs3)

    return (auc1 + auc2 + auc3)/3


start_time = time.time()

X = data.drop(['AktivEtterPassiv'], axis=1)
y = data['AktivEtterPassiv']

X_train, X_test, y_train, y_test = train_test_split(  X, y, test_size=0.25, random_state=0)

study = optuna.create_study(direction="maximize", study_name="LGBM Classifier")
func = lambda trial: objective(trial, X_train, y_train)
study.optimize(func, n_trials=100)

end_time = time.time()

print(f"The execution time is: {end_time-start_time}")

print(f"\tBest value (Score): {study.best_value:.5f}")
print(f"\tBest params:")

for key, value in study.best_params.items():
    print(f"\t\t{key}: {value}")

optuna.visualization.plot_param_importances(study)
```

```
.write_image("Bayesian_optimization_LR_importance.pdf")
optuna.visualization.plot_optimization_history(study)
.write_image("Bayesian_optimization_LR_history.pdf")
optuna.visualization.plot_slice(study, params=['C'])
.write_image("Bayesian_optimization_LR_C.pdf")



#Finding optimal threshold
mattbcc=[]

x=[]

for i in range(1000):
    threshold=i/1000

    x.append(threshold)

    y_pred1 = (probs1 >= threshold).astype(int)
    y_pred2 = (probs2 >= threshold).astype(int)
    y_pred3 = (probs3 >= threshold).astype(int)


    mattbcc.append(matthews_corrcoef(y_1, y_pred1)+matthews_corrcoef(y_2, y_pred2)+matthews_corrco


plt.plot(x, mattbcc, color='r')

# Naming the x-axis, y-axis and the whole graph
plt.xlabel("Threshold")
plt.ylabel("Objective value")

plt.savefig("mattbcc_lr.pdf")

plt.show()

mattbcc.index(max(mattbcc))/1000
```

G.4 Gradient Boosted Decision Trees Tuning

This section is built on the same principles as Logistic Regression, and it will thus only contain the hyperparameters tuned.

```
    param = {
        # "device_type": trial.suggest_categorical("device_type", ['gpu']),
        "n_estimators": trial.suggest_int("n_estimators", 10,3000),
        "learning_rate": trial.suggest_loguniform("learning_rate", 0.001, 0.3),
        "num_leaves": trial.suggest_int("num_leaves", 10, 5000),
        "max_depth": trial.suggest_int("max_depth", 3, 20),
        "min_child_samples": trial.suggest_int("min_child_samples", 0, 300),
        "reg_alpha": trial.suggest_loguniform("reg_alpha", 0.0000000000000000001, 50),
        "reg_lambda": trial.suggest_loguniform("reg_lambda", 0.0000000000000000001, 50),
        "min_split_gain": trial.suggest_float("min_split_gain", 0.01, 1),
        "subsample": trial.suggest_float("subsample", 0.01, 1),
        #"bagging_freq": trial.suggest_categorical("bagging_freq", [1]),
```

```
        "colsample_bytree": trial.suggest_float("colsample_bytree", 0.01, 1),
        'verbose': -1,
    }
```

G.5 Deep Learning Tuning

This section is built on the same principles as Logistic Regression, and it will thus only contain the hyperparameters tuned.

```
    num_layers = trial.suggest_int("num_layers",2,20)
    neurons_per_layer = trial.suggest_int("neurons_per_layer",2,200)



    param = {
        'activation' :
        trial.suggest_categorical("activation", ["identity", 'logistic', 'tanh','relu']),
        "alpha": trial.suggest_loguniform("alpha", 0.00000001, 1),
        "learning_rate_init": trial.suggest_loguniform("learning_rate_init", 0.00001, 1),
        "tol": trial.suggest_loguniform("tol", 0.000001, 0.1),
        "max_iter": trial.suggest_int("max_iter", 2,100),
        "hidden_layer_sizes": list(np.ones((num_layers),dtype=int)*neurons_per_layer)

    }
```

Norwegian University of
Science and Technology