Erasmus University Rotterdam

Master's Thesis
Computational Economics

Web-shop Order Prediction Using Machine Learning

Author:
Walter Hop

Student number:
316457

Supervisor:
Dr. Michel van de Velden

Co-reader:
Pieter Schoonees MSc

July 23, 2013

Summary

In the prudsys Data Mining Cup 2013, teams build classifiers to predict whether a web-shop user session will end in an order.

We apply a knowledge discovery framework, review various approaches for dealing with missing data, evaluate prediction models, and finally build a stacking classifier consisting of support vector machine, Random Forest and neural network base classifiers using the R environment.

Of the base classifiers, Random Forest provided the highest accuracy, followed by support vector machines. The neural network did not perform adequately. Besides having a high test accuracy of 90.3% on our prediction problem, Random Forest is shown to have many other desirable properties.

A custom implementation of stacked generalization proved to bring no accuracy advantages over Random Forest on our data set. This was true for a stacking classifier based on an ordinary least squares linear model, as well as one based on support vector regression.

In handling of missing data, we found no empirical differences in classifier accuracy when using various imputation methods.

Our classifier's prediction accuracy ranked 27th out of 63 competitors in the DMC competition. We assess the results in regards to other work, and discuss avenues and recommendations for future studies.

Keywords

machine learning; e-commerce; imputation; stacked generalization; Random Forest; support vector machines; neural networks

Contents

Chapter 1

Research problem

1.1 Introduction

In recent years, the shift in commerce from retail to web-shops has accelerated, with web-shops displacing physical stores quickly. E-commerce provides merchants with new opportunities for reaching out to prospective buyers at a large scale. E-commerce also brings new challenges. Relationships between merchant and consumer have become more volatile. No longer does a merchant know the customer, their wishes and intents directly. It is therefore much harder to incentivize customers through a personalized offering.

To mitigate this problem, a merchant is able to customize the content of their web-shop with respect to the visitor. Like never before, customers are leaving extensive trails of their online behavior. Large amounts of data are collected during each visit to a web-shop. This data can be linked to user information from a CRM database and common behavioral patterns of earlier visitors.

There is an obvious role for machine learning algorithms in analyzing the large amounts of data, picking useful attributes, and predicting future behavior of a visitor. When applied properly, personalized e-commerce may give a merchant a significant competitive advantage by increasing turnover and customer satisfaction.

The prudsys Data Mining Cup (DMC) competition [1] is a yearly competition, in which student teams work on a data mining problem. The competition of 2013 is centered around log data from a web-shop. The job of the data mining teams is to predict the probability that a certain visitor will place an order, based on user information and data collected during the visit.

Order prediction techniques can be used to personalize the experience of a web-shop. For instance, if a visitor is predicted to have a low order probability, the visitor might be motivated to buy by offering discount coupons. Visitors with a high order probability on the other hand, might be induced to purchase additional or more expensive products by advertising recommended products or product combinations. [2]

Currently, advanced machine learning capabilities for e-commerce are mostly restricted to expensive proprietary systems. This makes it interesting to build state-of-the-art prediction systems using open source components. The competition of the DMC allows for a quantitative evaluation of the solutions submitted by many university teams. Dissemination of the winning solutions will stimulate the development and use of machine learning solutions in e-commerce enterprises of all sizes.

1.2 Problem definition

The problem of the prudsys Data Mining Cup (DMC) competition in 2013 [1] is centered around log data from a web-shop. This data has been collected from actual web-shop visitors. Some of these visitors have placed an order at the web-shop; some have not. The goal is to learn from this data, and predict the probability that a new visitor will place an order. The size of the data is such (over 400,000 observations) that manual review is not realistic, requiring statistical or machine learning algorithms to summarize it.

When a user visits the web-shop, he or she may perform various *transactions*, each transaction becoming an observation in the data set. A transaction is formed by a series of page views and visitor actions during a time frame. Aggregated data for each transaction is available, such as: number of products clicked, highest price of products clicked, availability of the products, day and time, et cetera. Furthermore, the transaction is linked to some user data, such as customer age, type (business or consumer), gender, account lifetime, rating, and number of past payments.

One or more of these transactions by a unique user form a *session*. A session may, or may not, end in the purchase of a product. The core of the competition

is to build a prediction model that is able to predict whether or not an order will be placed in the session, based on transaction data. The output of a session's prediction should be an estimate of the order probability as a number between 0 and 1. The session is only described by its various transactions occurring over time, which means that part of the task is to aggregate the transaction data into useful session information.

In many analyses of real-life data sets, quality and completeness of the data constitutes a serious challenge. Collected data may come from various sources and may be incomplete. For instance, web-shop visitors may not have a registered user account, or may not have indicated certain preferences. As a result, the DMC transaction data contains many missing values, which requires a resolution strategy that introduces the lowest inaccuracies.

The DMC challenge is broken down into two tasks. We have provided a solution for the first task, which is an *offline classification* problem. For this task, DMC supplies a training set (with order outcomes) and a test set (without outcomes). The solution must be built using the training set, and is then executed on the test set to generate predictions. The algorithm may take a long time to calculate a prediction; therefore, competitors can use a multitude of complex methods. The second task is an *online classification* task, where a Java-based agent is to be built that provides a stream of classifications in a real-time scenario with very limited computing resources. Due to time constraints, we did not pursue the online task at this time.

The offline task is to be solved by creating a 'state-of-the-art' prediction system, or *classifier*, that can utilize large amounts of computational resources to make the best possible predictions. There are various prediction algorithms, or *prediction models*, that can be used for such a task. While some types of models generally give a good prediction accuracy, it is hard to determine in advance which type of model will perform well on a given data set. Fortunately, computing advances make it possible to create and evaluate various algorithms on a data set. A metric is necessary to estimate the accuracy of these models. This metric can be used to select the best performing model.

Each of these algorithms however, may have inherent biases and imperfections; for instance, some models may not be able to accurately recognize

high-dimensional patterns in the data. Therefore, it is interesting to investigate if a classifier can be created that builds on the strengths of diverse models, in order to generate even more accurate predictions. Such a classifier, which learns from the output of other classifiers, is called a *meta-classifier*.

1.3 Research questions

The main question we aim to answer is the following:

- How can we predict the probability that a web-shop visitor will place an order, based on transaction data?

Subquestions that arise and need to be resolved in order to provide a solution for this question are:

- How can we create a prediction system using only open-source components, to extract useful features from web-shop transaction logs, build prediction models, and generate predictions?
- What is a good strategy to deal with large numbers of missing values in prediction model training?
- Can we construct a meta-classifier that builds on the output of various prediction models, in order to make even better predictions?

Chapter 2

Methods

2.1 Knowledge discovery framework

This chapter will describe the various steps of the analysis and the techniques used. We will follow the framework for knowledge discovery in databases (KDD) tasks described by Tan et al. [3]:

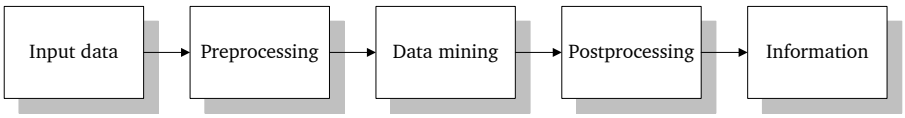


Figure 2.1: Framework for knowledge discovery in databases

For data mining tasks, the *input data* is usually large in size, i.e. it consists of many observations, many variables, or both. The relationships between observations and attributes are often not known. Sometimes, the data is in a format that does not immediately yield to numerical analysis, for instance, the input data might consist of camera images. In most cases, the data in its current form will not be able to answer the research question.

A *preprocessing* step readies the raw data for efficient analysis. This step might include various necessary processes, such as *feature extraction*, i.e. the generation of useful continuous or discrete variables from the input; *dimensionality reduction*, or the transformation of high-dimensional data into a lower number of new variables; *feature selection*, or throwing away uninformative variables to improve accuracy and performance while retaining others in their original dimensions and units; *normalization*, or the transformation of data to zero mean and unit

variance for more unbiased analysis; or *subsetting*. Other useful processes not mentioned by Tan et al. could be data cleaning, combining data from several sources, and *imputation*, the reconstruction of incomplete input data for later use with numerical or statistical algorithms that require full data matrices.

Data mining can broadly be divided into two categories. One is *descriptive modeling*, where the goal is to uncover relationships, patterns, and clusters present in the data. The other is *predictive modeling*, where the goal is to predict some attribute using other attributes. Prediction can either take the form of *classification* where an observation is predicted to be a member of two or more pre-defined sets or classes, or *regression* where a continuous variable is estimated.

Finally, the results from the data mining step can optionally be *postprocessed*. The output is often visualized, interpreted, transformed, or filtered to focus on areas of interest.

The result of the process is *information*. The key difference between information and raw data is that the information is structured, summarized, and immediately useful for decision making.

2.2 Overview of study design

This section will broadly describe our study design in terms of the KDD framework. The study is predictive in nature; that is, it will yield a single prediction for each visitor session, based on a set of features. The predicted attribute is a binary variable: the visitor is predicted either to place an order, or place no order. Thus, the task is a *binary classification* problem.

The main structure of the process is presented in Figure 2.2. This section briefly lists the various steps taken:

- **Input data:** The raw transaction data are collected and parsed, resulting in a training set (data and their observed outcomes: order or no order) and a test set for which the outcomes are not known.
- **Feature extraction:** The transaction data is aggregated into session data, creating a number of possibly informative features, i.e. variables describing the session. These variables are used for building prediction models and generating predictions.

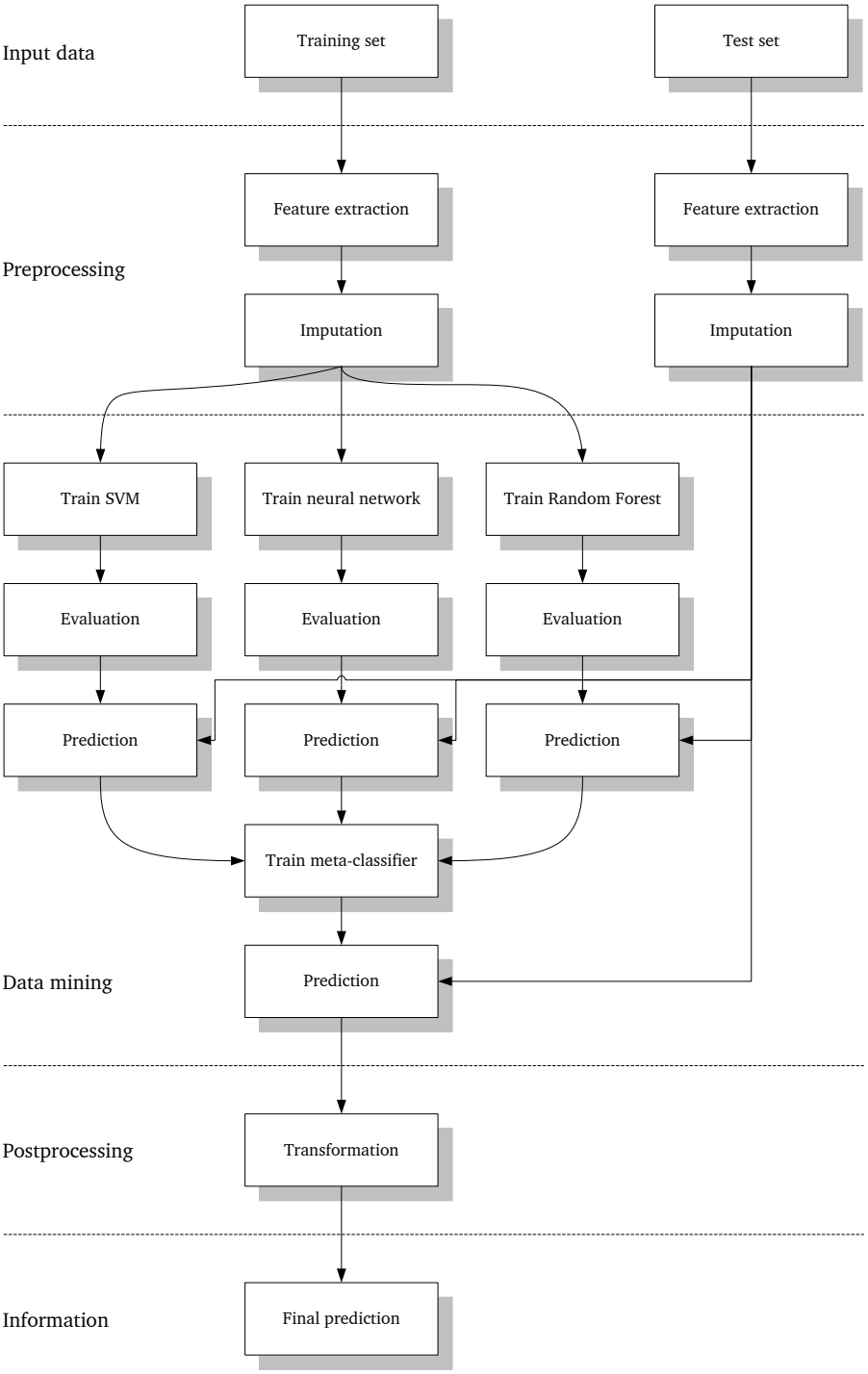


Figure 2.2: Detailed analysis process

- **Imputation:** Incomplete observations, for which not all features are known, are processed to create full data matrices amenable to further analysis.
- **Training:** The imputed training set is used as input for various prediction algorithms, to create various prediction models or classifiers.
- **Evaluation:** The expected accuracy of the models is estimated. This estimation is used to *tune* the models for the highest possible accuracy.
- **Prediction:** The models are executed on the preprocessed test set, yielding a prediction for every session in the test set.
- **Meta-classifier training:** We attempt to improve on the individual models by building a meta-classifier that uses the output of component models as its input.
- **Meta-classifier prediction:** The meta-classifier is executed on the test set, yielding final predictions.
- **Transformation:** The prediction output is transformed into a format that more optimally resolves the stated optimization goal.
- **Final prediction:** The produced information resolves the original prediction problem.

Process steps and algorithms requiring further introduction are extensively described in the following sections.

2.3 Input data

One visit by a user to the web-shop is called a *session*. During this session, the user might click on various products and possibly add these to their shopping basket. Finally, the user may choose to order some products from the web shop.

Each session is divided into one or more *transactions*. The data about these transactions is provided by DMC [2]. We list the variables given in Table 2.1. For each variable, we provide the description, range, and whether the column can have missing values (NA or Not Available).

Variable	Description	Values	NAs?
sessionNo*	Running number of the session	int	no
startHour*	Hour in which session has begun	0–23	no
startWeekday*	Day of week in which session has begun (Mon–Sun)	1–7	no
duration	Time in seconds passed since start of session	float	no
cCount	Number of the products clicked on	int	no
cMinPrice	Lowest price of a product clicked on	float	yes
cMaxPrice	Highest price of a product clicked on	float	yes
cSumPrice	Sum of the prices of all products clicked on	float	yes
bCount	Number of the products put in shopping basket	int	no
bMinPrice	Lowest price of a product put in shopping basket	float	yes
bMaxPrice	Highest price of a product put in shopping basket	float	yes
bSumPrice	Sum of the prices of all products put in basket	float	yes
bStep	Current purchase processing step	1–5	yes
onlineStatus	Is the customer on-line?	y, n	yes
availability	Delivery availability of the product(s)	string	yes
customerID*	Customer number	string	yes
maxVal*	Maximum admissible purchase price for the customer	int	yes
customerScore*	Customer rating from the point of view of the shop	int	yes
accountLifetime*	Lifetime of the customer account in months	int	yes
payments*	Number of payments made by the customer in the past	int	yes
age*	Age of the customer	int	yes
address*	Company/Gender (1=Mr, 2=Mrs, 3=Company)	1–3	yes
lastOrder*	Time in days passed since the last order	int	yes
order*	Did the customer order a product in this session?	y, n	no

Table 2.1: Input variables describing a web-shop transaction. [2] Each transaction is part of a session, identified by *sessionNo*. Variables marked with * are constant throughout all transactions of a single session. Numbers are given as integers (int) or floating point values (float).

2.4 Preprocessing

At the end of preprocessing, we will choose an approach to deal with missing values in the data. There are many possible strategies to handle missing values, but they may differ in the quality of the resulting data.

2.4.1 Excluding missing values

A simple way to handle missing values during classifier training is to disregard any incomplete observations and only include the subset of complete examples in the model. When encountering missing values in a test observation during prediction, one might produce a 'default prediction', such as a predicted order probability of $p = 0.5$ or the expected probability $E[p]$ as learned from the training set.

This approach might be interesting in cases where the data is plentiful and missing values appear only in a reasonably small fraction of the observations. However, the approach assumes that the occurrence of missing values itself is not correlated with the outcome. If there is a strong correlation, the subset of remaining training data will not be representative, and the model may perform badly.

2.4.2 Mean imputation

During *imputation*, each missing value is replaced by a discrete or continuous value. An important advantage of imputation over missing value exclusion is that no subsetting takes place, which allows all observations to be used for analysis.

Mean imputation is a simple procedure which for every variable determines the mean (or for categorical variables, the mode) across all training observations, and substitutes any missing values with this value. For instance, if the price of a product viewed is unknown, the average product price can be assumed.

The mean-value approach may be attractive if the resulting data set is used for calculations. However, replacing many missing values by any single estimate might affect the structure of the data, as it could create a cluster of observations around the imputed value. Imputing the mean may also create the appearance

of a stronger relationship with other variables than is warranted, though this may not necessarily be a problem for our specific application.

2.4.3 Predictive imputation

In predictive imputation models, a missing value is *predicted* using the other variables as predictors. For every feature that has missing values, a prediction model is created. As a prediction algorithm, we may use any model, such as the models described in section 2.5. (Note that strictly speaking, taking the mean of a variable may also be construed as a predictive operation, but for our study we will treat it separately.)

The subset of observations that have an actual value for the variable are used as the training set. After model training, predictions are generated for all the observations that have a missing value for the variable.

For many purposes, prediction may be a more useful choice than the often-used mean-value imputation, as it produces a distinct 'plausible' value for every observation, which may be more accurate than assuming the mean. A drawback of predictive imputation however is the added computational effort required to fit prediction models for every variable.

2.4.4 Unique-value imputation

A technique that seems to be rarely used is *unique-value imputation*, where missing values are simply replaced by a value that is unique for that data set. For instance, for categorical variables, we could create a new dummy category 'missing', and for numerical variables, we could insert a unique number. To give an example, we might either replace all missing numbers with the value –1, or we might give all missing values distinct values, such as –1, –2, et cetera.

This approach would be highly inappropriate if the result would be used in calculations, for instance for summarizing the data. The imputed values distort the distributions of the variable and have no semantic value in the problem domain. For instance, imputing a 'product price' variable with –1 makes it hard to later calculate the mean product price. It is unknown if this disadvantage of unique-value imputation is relevant to our main prediction problem. After all, we are only interested in predicting order probabilities. The actual values in the

training set are not directly relevant to us, as long as our classifier's prediction accuracy is not impaired. This can be determined empirically.

In our main problem, we use a prediction model to estimate the response variable, using all variables as predictors. This raises the question if the added effort of mean or predictive imputation is warranted at all. Saar-Tschehansky and Provost [4] discuss the concept of *imputability:* in a highly imputable variable, its missing values can be predicted well from other variables. If the imputability of a variable is *high*, and the value is a significant predictor for the outcome, it might follow that a classifier would likely be able to predict the response from the other variables directly without the help of an intermediate imputation step. If the imputability of the variable is *low*, apparently it is not strongly correlated with other variables, and imputation attempts might not be fruitful in any case.

We might be interested in unique-value imputation for various reasons. First, imputation methods which replace a missing value by the mean or another plausible number could destroy useful information. After applying mean imputation or predictive imputation, it is no longer known if the value was originally missing, which *might* actually be relevant. Second, imputing with a simple unique value is a very fast procedure, comparable to mean imputation, but retaining the 'missingness' of an observation's variables.

2.4.5 Imputation selection

For time-saving purposes, we prefer to choose a single imputation approach for the remainder of the analysis. To select the most appropriate imputation method, we will first execute all viable imputation algorithms on the data set. We will then train classifiers on each of the imputed training sets.

We will rank the imputation methods according to the estimated accuracy of the resulting classifiers. In this way, we will hopefully obtain a good estimate of their usefulness for our problem. The best imputation method will be selected for the remainder of the analysis.

See section 3.3 for the results of the various imputation approaches and the selection process.

2.5 Data mining

After selecting an imputation approach, we can use the imputed training set to train prediction models. In the next sections, we will discuss the prediction models used, as well as the methods used to compare and combine them.

2.5.1 Model evaluation

Overfitting and underfitting

A data analyst has a large number of prediction models at their disposal. Each type of prediction model may have imperfections and biases, and the relative accuracies of the model types may vary based on the data set at hand. We will use various prediction models which are all based on vastly different mathematical and statistical methods. Therefore, we must have a way to evaluate their relative accuracies on our data set empirically.

Since we have access to outcomes of only the training set, we can only use the training set to evaluate models. The best model is not necessarily the model that has the best classification accuracy on the training set itself — or the lowest *training error*. As an extreme example, a naive model could just store all training examples in memory and perform a simple lookup on prediction of a test observation. This model would have a 100% accuracy on the training set, but it would be useless in practice, since it would fail on unseen observations having just slight differences. Thus, this naive model would have a high *test error*.

What we are looking for is a prediction model that *generalizes* well to new observations, minimizing the test error. A model generalizes well if it recognizes and remembers the most significant patterns in the training set, without getting disturbed by insignificant noise and randomness in the data.

Good generalization can be viewed as finding the balance between two pathological extremes. One extreme is *overfitting*, i.e. optimizing the model too far on the training set with its noise and random patterns. As our example of the 'naive model' shows, an overfit model has a deceivingly low training error, and would likely have a higher error when making predictions on unseen observations. At the other end, there is *underfitting*, where the model contains

too little information about underlying patterns to make correct predictions. [3: 4.4]

Estimating the test error

We must optimize, or *tune*, each prediction model for the best generalization capability, or the lowest test error. There are various techniques to assess this model property. Subsampling methods, such as *cross-validation* and *bootstrapping* are often used for this. The general idea of subsampling methods is that a model is trained on only a subset of the training set. The remainder of the data can then be used to test the model.

In k-fold cross-validation, the training set is split into k subsets which should be roughly of equal size. For each group or fold, the model is trained using the complement of the fold and then tested on the fold itself. For example, in 10-fold cross-validation, the model is trained and tested ten times, each using a different 90-percent chunk as a training set and the 10-percent held-out data as a test set. Each of these runs produces a test error, which functions as a fair estimate of how the particular model with this settings would generalize to unseen observations. The average of these test errors, the *cross-validated error estimate*, as well as its variance over the different folds, are noted. After the k training subsets have all been tested, the model is trained on the full training set.

Bootstrapping follows a similar procedure. In bootstrapping, a training subset is created by taking a *bootstrap sample*, i.e. observations are drawn from the original training set *with replacement*. This leads to a training subset that can potentially contain multiple copies of observations. The full data set is then used as a test set, leading to a test error estimate. This procedure is then repeated a number of times; for instance 25 times, which is the default in the R caret library. [5] The duplication of observations in training subsets can lead to low error estimates. There are remedies for this, such as the *.632+ bootstrap*, which applies a correction. An advantage of the bootstrap is that it produces smoother estimates, i.e. the estimates have less variability when compared to cross-validation. [6]

For most purposes, bootstrapping and cross-validation produce comparable results, however we choose to perform cross-validation, as the bootstrap estimate

can break down in some overfit situations [7: 7.11], and cross-validation is a more simple procedure to implement.

It is important to realize that these techniques cannot *conclusively* determine the value of the test error, since that would require access to the actual classifications of the test set, while only the training set is available. However, they can provide an estimate of the *expected* test error. [7: 7.12] While this is the best estimate we can make, we must remain vigilant for any surprises that may arise in practice.

Tuning a model

For each model, there are one or more *hyperparameters*, i.e. parameters on model level, that affect the internal structure of the model and the way it is trained. For instance, in a neural network, we can choose the number of internal nodes. These hyperparameters often affect the accuracy of the model; so, we must search for the optimal set of hyperparameters. We can do this by training multiple models and comparing their cross-validated error estimates. This is called *tuning* the model or tuning the model parameters.

For the DMC problem, we select the best model by minimizing two distinct error estimates: first, the default *binary test error* when giving binary class predictions (i.e. only 'order' or 'no order'); second, the *DMC test error* which uses an error function based on predicted probabilities as specified by DMC [2] and as given in Equation 2.1. By considering the linear difference between observed outcome (a number 0 or 1) and the predicted probability, the DMC error takes into account that we can submit as predictions a *probability* that an order will be placed, i.e. any value from 0 to 1.

$$err_{DMC} = \sum_i |order_i - prediction_i| \tag{2.1}$$

Note that the DMC error is a sum of errors for all observations; therefore it also depends on the number of observations. For classifiers generating only binary predictions, the DMC error is the total number of errors on all test observations. We will provide both binary and DMC error where appropriate.

Usually, accuracies are given as fractional values between 0 and 1. To make meaningful comparisons between the two error estimates, we define the *DMC*

accuracy measure so that, for instance, a mean err_{DMC} per observation of 0.3 will lead to an acc_{DMC} of 0.7:

$$acc_{DMC} = \frac{n - err_{DMC}}{n} \tag{2.2}$$

With the evaluation methods outlined, we will now discuss the prediction models used. For the results of model evaluation when applied to the various models, see section 3.4.4.

2.5.2 Random Forest

Random Forest is a popular and versatile prediction model. The Random Forest algorithm is based on *decision trees*. A decision tree is an older prediction model. To understand Random Forest, we must first cover decision trees.

Decision trees

A decision tree is created from training data by iteratively *splitting* the training observations into subgroups or tree branches. For each internal tree node, a split condition is created on the value of a feature (for instance $age > 50$). The algorithm tries to find the split that creates the most 'pure' subdivisions of the data, i.e. the split leads to subgroups that best differentiate the observations into classes. [3: 4.3.2] There are many algorithms for growing a decision tree, such as ID3, C4.5, C5.0, CART, and MARS. [7: 9.2]

We will illustrate the outcome by building a decision tree using the R tree package [8]. As a training set, we use Fisher's well-known Iris data set [9], which contains measurements on various *Iris* flowers, as well as their species (setosa, versicolor, or virginica). For clarity and easier plotting, we will only use two features present in the data set: the petal length and petal width. We will use these features to predict a flower's species.

As can be seen in Figure 2.3, the decision tree algorithm produces a model for the data that is simple and easy to interpret. Making a prediction for a new observation from the model is done by starting at the root of the tree, then iteratively moving down the branch that matches the features of the observation. Ultimately, we end up at a single leaf node of the tree. The leaf node is labeled with the class that we will predict.

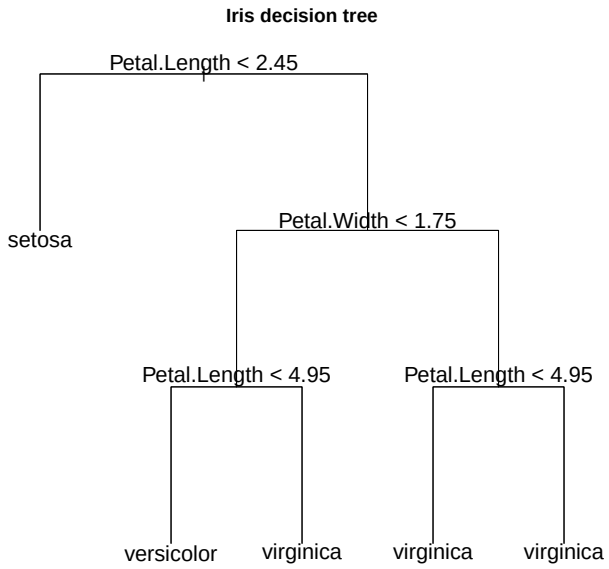


Figure 2.3: A single decision tree for the well-known Iris data. The features 'petal length' and 'petal width' are used to predict the species of an Iris flower.

The simplicity of a decision tree also brings disadvantages. A single decision tree is not useful for modeling complex patterns, since the partitions created by the tree splits are always *parallel* to the axes in feature space. An example of this is shown by plotting the decision tree's splits in Figure 2.4. The decision tree functions well on the Iris data, as its classes are separated by the orthogonal lines very well, but this does not hold for many other data sets. Decision trees are also known to be *unstable*, as they are sensitive to noise in the data. [7: 15.2] This means that small, insignificant changes in the training data can lead to different decision trees and prediction results, which is undesirable.

The Random Forest ensemble

The Random Forest approach improves on the stability and accuracy of decision trees by embedding a large number of decision trees in a so-called *ensemble classifier*. An example Random Forest, for instance, might contain 500 decision trees. Every decision tree is trained on a bootstrap sample from the training set. (Review section 2.5.1 for details on bootstrapping.) A prediction is obtained by having all decision trees create a prediction, and taking the average or a majority

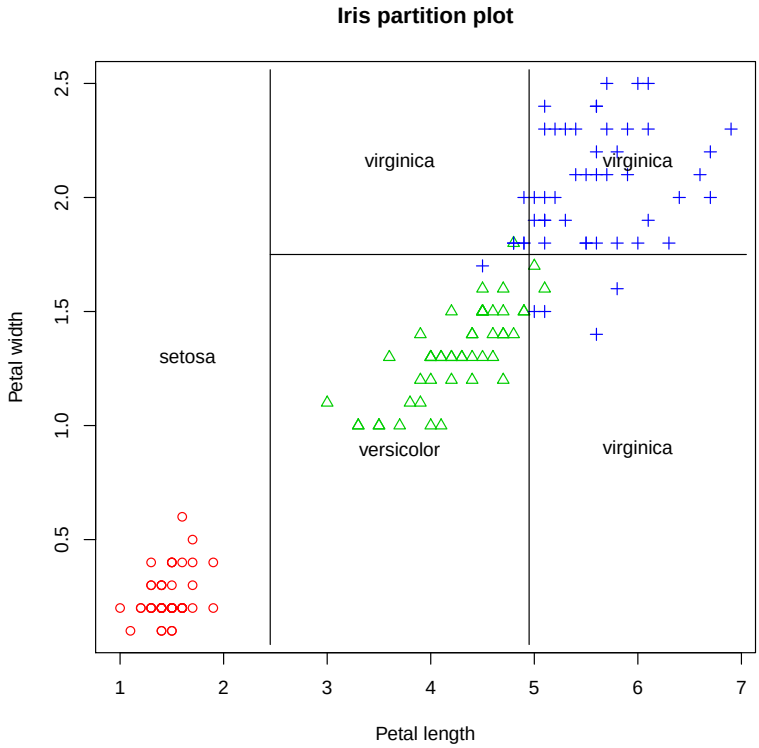


Figure 2.4: Partition plot for the Iris decision tree from Figure 2.3. Observations are plotted as points. Actual classes: red circles: setosa, green triangles: versicolor, blue crosses: virginica. The tree model is superimposed. Each split, or internal tree node, corresponds to a horizontal or vertical line in feature space. These partition lines form the decision boundaries. Predicted species are mentioned in text within their boundaries.

vote. Applying this procedure to decision models is called *bagging* (short for *bootstrap aggregating*). It can lead to a higher stability and improved accuracy. [7: 8.7.1]

Random Forest further improves on bagging by 'de-correlating' the trees. This is accomplished by considering only a small, random, subset of features in each tree split. [7: 15.2] If there are many features in the data set, this constraint ensures that individual decision trees look vastly different from each other. This brings an accuracy improvement which compares favorably to *boosting*. [10] In boosting, various individual models are also used, but the contributions of models

and training observations are weighted iteratively, in order to improve accuracy of the ensemble. [7: 10.1] Boosting, while powerful, can be a computationally expensive procedure. Random Forest on the other hand is much faster, with comparable accuracy. The random feature selection also makes Random Forest well suited for data sets with many features and missing values. [10]

The algorithm for creating a Random Forest as used in the R randomForest package [11] is implemented as follows. A predefined number of decision trees is trained. For every tree, a bootstrap sample is taken from the training set. This tree is then trained on the bootstrap sample, choosing at every split only from a fixed number of randomly chosen features. Predictions can be made from the Random Forest by feeding a new test observation to all individual decision trees and then averaging their predictions or taking a majority vote.

Out-of-bag data

As each individual tree is trained only on a bootstrapped subset of the training data, a part of the original training data is held out for that particular tree. This is called the 'out-of-bag data'. [10] As each bootstrap sample contains duplicate observations, many training observations remain unused in that sample, so each training observation has a chance of around 36% to be part of the out-of-bag data for a certain tree. [11]

Random Forest uses the out-of-bag data to compute two measures which make it very fast for practical analysis. The first measure is the *out-of-bag error*. The out-of-bag error for a single tree is constructed after training of a tree by feeding all out-of-bag data to the tree for prediction, and recording the mean error between the predicted outcomes and actual outcomes. This error is then averaged over all decision trees. The out-of-bag error provides a quick estimate for the test error, without the computational effort of cross-validation.

The second internal measure provided by out-of-bag data is the *variable importance* of each feature. In Random Forests, there are many possible algorithms for calculating this measure. [12] It is derived in a similar way to the out-of-bag error. In one scheme, the importance of a feature is determined by first randomly permuting the out-of-bag data for that single feature, then generating predictions with the permuted data, and finally recording the *mean decrease in accuracy* when

comparing the normal out-of-bag error and the permuted out-of-bag error. [10] In the permuted data set, the tested variable is essentially random noise, so the absence of its relationship with the outcome provides a good estimate for the contribution of that variable to successful prediction. In other models, feature importance ranking is usually an expensive process, often done by step-wise feature selection and repeated re-training of the complete model. Random Forest provides this information without that large computational cost.

However, Random Forest importance measures cannot always be relied upon if features vary strongly in their scale of measurement or number of categories. This can lead to cases where importance measures are biased and have unnecessarily high variance. Strobl et al. [12] note that this problem is inherent to decision trees and bootstrapping, and instead suggest changes to tree building algorithms and the subsampling method.

Note also that Random Forest only looks at univariate (single-variable) importance. It does not take into account multivariate patterns, such as variables which are collinear or otherwise highly correlated. Random Forest considers a random subset of variables for each tree split, and thus variables behaving similarly might all be used often in tree splits and thus jointly receive a high importance estimate, while for analysis some of the variables may still be redundant.

Hyperparameters

It has been suggested that Random Forests do not suffer from overfitting. [10] While this is true for many data sets, there are some cases where overfitting can happen [13], so tuning of the model for best generalization capability is advised. A Random Forest has only few hyperparameters to be defined, which makes this process relatively fast.

One hyperparameter, the number of features considered randomly at each tree split, often called m_{try}, must be selected before training starts. The other hyperparameter, the number of trees or n_{tree}, can be set to an arbitrary large value such as 500, since it has no local optimum: at some point, the accuracy will simply no longer increase by adding more trees.

Random Forest accuracy is affected by two properties: a higher *correlation* between any two trees in the forest increases the error rate; and a higher *strength*

(or accuracy) of individual trees in the forest decreases the error rate. m_{try} affects both of these properties: a lower value of m_{try} lowers both correlation and strength, and a higher value of m_{try} raises correlation and strength. [10]

The competing forces of correlation and strength affect the test error inversely. Therefore, we must search for the optimal test error by varying m_{try}. We train Random Forests on various values of m_{try}, and compare the estimated test errors of all Random Forests using the out-of-bag or cross-validated error estimates.

For the results of Random Forest training and tuning on our data set, see section 3.4.1.

2.5.3 Support vector machines

Support vector machines (SVMs) are a family of powerful prediction models based on statistical learning theory. SVMs are suitable for recognizing complex patterns in high-dimensional feature sets.

Maximum-margin hyperplane

A support vector machine divides the training set's observations into two groups by finding a *separating hyperplane* between the groups. Unlike a decision tree, where each split adds a new hyperplane that divides the remainder of feature space (recall Figure 2.4), the SVM yields only one single hyperplane. In a two-dimensional data set, this hyperplane can simply be a line that divides two clusters of data. (An SVM can also find more complex separation boundaries, which we will see shortly.)

In many cases, there are infinitely many possible hyperplanes that separate two classes. However, in contrast to most other models, an SVM always finds the *maximum-margin hyperplane*. This is the separating hyperplane that has the largest distance to any data point in the training set. This is illustrated in Figure 2.5 by line B_2, which has a larger margin than the other two lines.

All separating hyperplanes produce the same training error, meaning they separate the same points from the training set correctly. Choosing the maximum-margin hyperplane however tends to create a better generalizing classifier, and therefore a lower *test error*. If the margin is small, it follows that a small change to a data point may easily cause it to move over to the other side of the hyperplane,

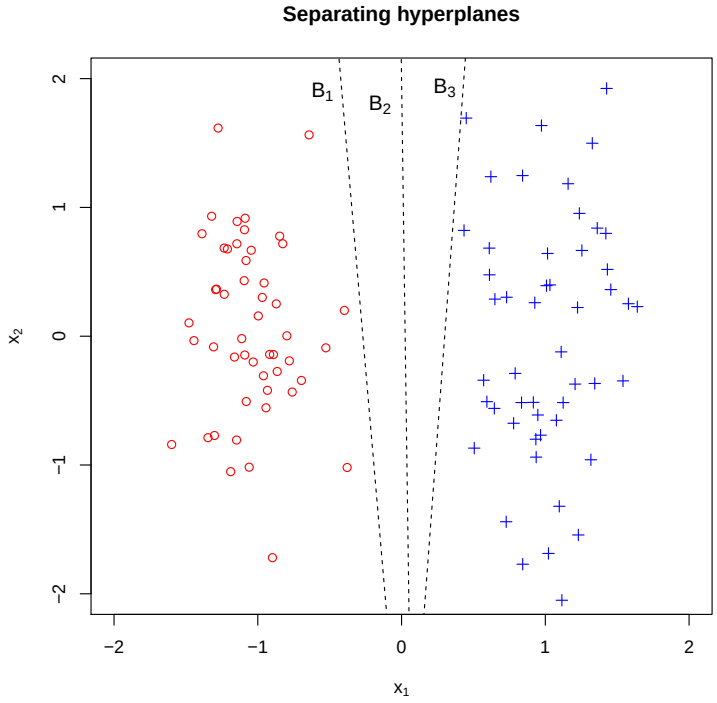


Figure 2.5: Many hyperplanes separate these two classes. The maximum-margin hyperplane (shown as B_2) has the largest distance to any point.

causing a change in the model's prediction. A small margin makes a classifier prone to overfitting, and causes it to generalize poorly to new data points. [3: 5.5.1]

The data points which lie nearest to the hyperplane, on the edge of the margins, are called the *support vectors*. A model created by an SVM can be described fully by its support vectors. The number of support vectors in the model is a measure for the complexity of the decision surface.

Finding the maximum-margin hyperplane is a minimization problem that can be solved iteratively through numerical methods. This unfortunately is a slow process if the training set contains a large number of observations (however, the method is less sensitive to the number of features). Modern SVM implementations may use varying efficiency optimizations to find a faster solution, [14] so the computation time also depends on the software used.

With its single hyperplane, an SVM functions as a *binary* classifier, always

dividing observations into two classes. Prediction for an observation can be performed by calculating on which side of the hyperplane the point lies.

Multi-class prediction is often accomplished by creating an SVM for each pair of classes in a 'one-against-one' fashion (finding the hyperplane that separates classes c_i and c_j). Alternatively, a SVM can be created for every class in a 'one-against-rest' approach (finding the hyperplane between class c_i and its complement $\neg c_i$). [3: 5.8] The class prediction is then determined by aggregation of the predictions of all SVMs, for instance by majority voting.

Nonseparable problems

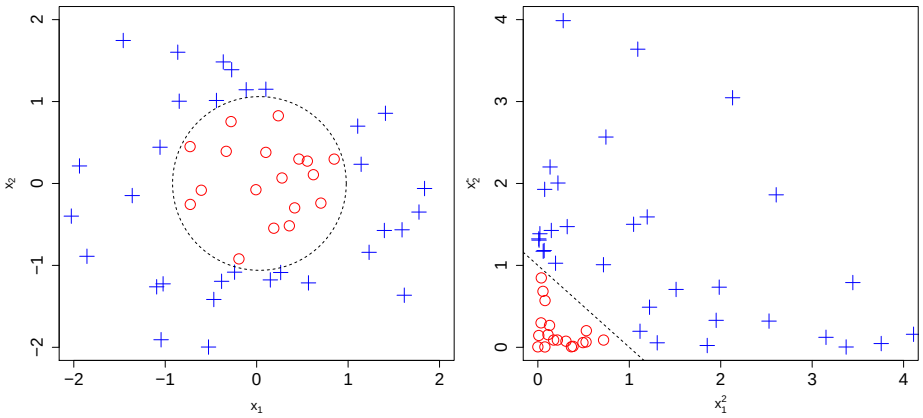
In the case that no hyperplane can be found that correctly separates *all* training data points, the data set is called *nonseparable*. In this case, the SVM algorithm can still find a good hyperplane with a margin that is somewhat 'soft' on both sides, meaning that it allows for some misclassifications. This is achieved by introducing slack variables into its system of equations. Observations on the wrong side of the margin are weighted down in order to reduce their influence. [15] This mechanism is known as the *soft-margin approach*. [3: 5.5.3]

In this setting, there are now two goals to optimize for: first, the *margin* must be maximized, but second, the *slack variables* should be minimized. If we do not minimize the slack variables, we might arrive at a wide-margin classifier that has a high number of misclassifications. The balance is influenced by the *cost parameter* C which must be set before SVM training begins.

Nonlinear problems

Many data sets are not linearly separable in their original dimensions of feature space, as they contain nonlinear patterns. This is illustrated in Figure 2.6(a). Here we see an example data set that is not linearly separable in its feature dimensions. There is no straight line that we can possibly draw that would satisfactorily separate the data, even when allowing for some misclassifications through a soft margin.

However, if we would transform this data using a transformation function that maps points with features x_1 and x_2 to a transformed space with dimensions ${x_1}^2$ and ${x_2}^2$, we arrive at Figure 2.6(b). It turns out that in fact we *can* linearly



(a) Nonlinear separation in feature space. (b) Linear separation in transformed space.

Figure 2.6: The two-class data set in (a) is not linearly separable in the dimensions of its features x_1 and x_2. However, a circle defined by ${x_1}^2 + {x_2}^2 = 1$ would separate the classes. If we transform the data set to dimensions ${x_1}^2$ and ${x_2}^2$ as illustrated in (b), the data becomes linearly separable in the transformed space.

separate the data in these dimensions.

Nonlinear SVMs can exploit this property. The approach can be thought of as projecting the original data points into a higher-dimensional space, in order to find the linear separating hyperplane in that space. Unless the training set contains contradictions (e.g. two observations with equivalent features, but different classes), it is always possible to find a linear separation boundary in a higher dimension.

A question remains what kind of mapping function to use on a certain data set, in order to ensure that a separating hyperplane can be found. Also, finding numerical solutions in a very high-dimensional transformed space might look computationally intractable. However, fortunately it is not necessary for the SVM to actually carry out a full projection of the training set to a higher-dimensional space, or actually perform calculations in that higher dimension. To circumvent projection, SVMs employ a technique which is known as the *kernel trick*. The trick depends on a property of the SVM's numeric optimization process that finds the maximum-margin hyperplane. This process only requires the *dot product* of two data points to be known; besides the dot product, no access to actual values is necessary. The dot product functions here as a 'similarity' metric of two data

points. [3: 5.5.4]

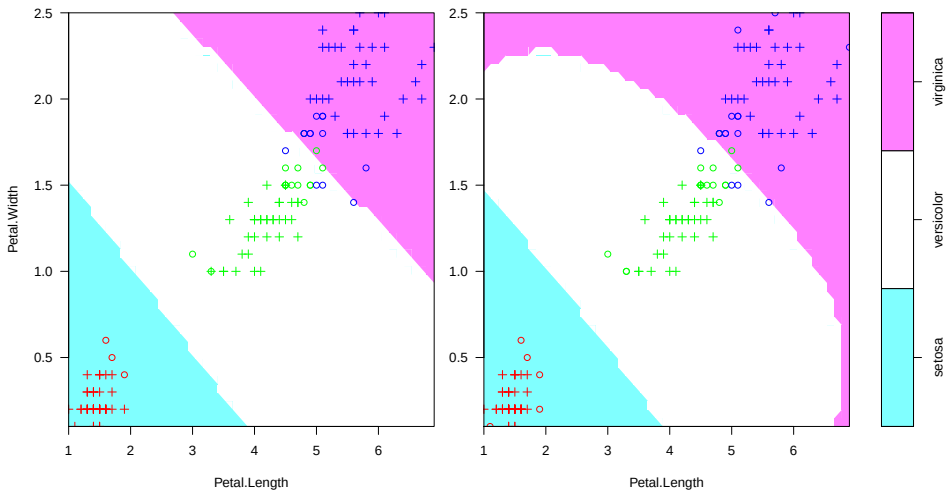
We can compute the similarity between two data points in a *transformed* space by using a *kernel function* applied to the data points in *original* feature space. The similarity function must be equivalent to computing actual dot products in the higher-dimensional space. There are various kernel functions available that satisfy this equivalence.

One of the possible kernel functions is the *polynomial kernel*. Going back to the example of Figure 2.6, the polynomial kernel can be an appropriate kernel function to use. If a SVM would use as its similarity function a quadratic (2-degree) polynomial kernel function applied to the data points in (a), the SVM will arrive at a solution that is linearly separable in a quadratic function of the data points, such as shown in (b). A 2-degree polynomial kernel SVM would therefore be powerful enough to separate the quadratic pattern in the data set.

Another well-known kernel is the *radial basis function* (RBF) kernel, which is equivalent to mapping the data points into an infinite-dimensional space. [16: 15.2.3] A linear hyperplane in infinite-dimensional space can translate to an extremely complex separation boundary in feature space. Therefore, this is a very powerful kernel that allows the SVM to generalize highly complex patterns.

However, as kernel functions become more complex, the computation times of SVM training will increase. So, in large data sets it might be better to use a relatively simple kernel, such as the *linear kernel*, which is a 1-degree polynomial kernel that can be computed relatively quickly. This of course runs the risk that complex class boundaries can not be separated, resulting in a higher training error.

We give an example of some typical separation surfaces created by support vector machines. We train two SVMs on the Iris data set [9]. (For an introduction of the Iris data, please review section 2.5.2.) We train one SVM using a linear kernel, one using a radial basis kernel. The results are plotted in Figure 2.7. The linear kernel produces linear decision boundaries. The radial basis kernel produces a more rounded boundary between classes ‘versicolor’ and ‘virginica’. Both kernels seem to provide a good separation for this prediction problem.



(a) Linear kernel SVM

(b) Radial basis kernel SVM

Figure 2.7: Prediction plots of two support vector machines trained on the Iris data set. Data points are colored with their actual class (red: setosa, green: versicolor, blue: virginica). Support vectors are marked as circles. Predicted classes are differentiated by background color.

Hyperparameters

We recall that a hyperparameter in soft-margin SVMs is C, or the *cost parameter*. This parameter sets the penalty for misclassifications. When C is increased, the SVM will try harder to prevent misclassified data points. This leads to a smaller margin [16: 15.2] and a more complex decision boundary [14], which has a hazard of overfitting. Decreasing C will lead to a smoother boundary as the SVM becomes lax in preventing misclassifications. At some point, this will lead to underfitting. Usually, there is an optimum for C.

Another important hyperparameter in nonlinear SVMs is the choice of kernel function as described in the previous section. Karatzoglou et al. [14] provide some guidance as to which kernel function is appropriate for a problem: they advocate the *radial basis (RBF) kernel* for general purposes, the *linear kernel* for large sparse problems such as text mining, the *polynomial kernel* for image processing, and the *sigmoid kernel* as a proxy for neural networks.

Some of these kernel functions have parameters of their own that must be set before training as well. These parameters influence the kernel function's

dimensionality or shape.

The polynomial kernel function accepts a *degree* parameter p that affects the order of the polynomial patterns recognized. If $p = 1$, the kernel is called a linear kernel, and its decision surfaces will also be linear in feature space. A large p allows separating more higher-order polynomial patterns in the data.

The RBF kernel has a γ (gamma) parameter that affects the *width* of the kernel. The choice of γ influences the shape that data points form in the transformed space, and thereby the shape and complexity of the recognizable decision boundaries in feature space. γ is sometimes called the 'inverse-width' parameter. A small γ creates a wide kernel, which will lead to more linear boundaries, while a large γ creates a narrow kernel that allows for more flexible boundaries. [17] If the data set is very dense, selecting a narrower kernel is preferred. If the data set is sparse, a wider kernel generally gives better results. [18] As is often the case, a highly complex decision boundary can be a sign of overfitting, where a too simple boundary might hint at underfitting.

The generalization capability of a SVM can be very sensitive with regards to the choice of C, the kernel function, and the kernel's parameters. [15] The optimal choices for C, kernel, and kernel parameters can be determined by repeatedly training SVM models in various configurations, and selecting the best performing model through a comparison of their cross-validated accuracy estimates. (Review section 2.5.1 for details about the cross-validation process.)

The results of the SVM tuning procedure will be described in section 3.4.2.

2.5.4 Neural networks

An *artificial neural network*, or shorter: neural network, is a parallel computing device modeled after the structure of biological neural networks, such as found in the human brain.

Neurons and weights

A neural network consists of many simple processing units, also referred to as *neurons* or nodes. Each node combines a number of incoming signals and produces an outgoing signal. The output of the node is determined by a *transfer function* that summarizes the weighted inputs, and an *activation function* that

produces the output value. Often, a node also uses a *bias*, also called threshold, which is an incoming signal with a constant value. An activation function often produces an output signal in the range $[0, 1]$ or $[-1, 1]$. The output signal might be an input for another neuron. A schematic diagram of a neuron is presented in Figure 2.8.

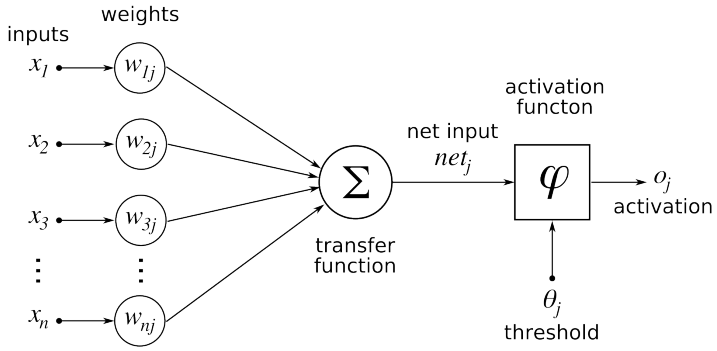


Figure 2.8: Model of a neuron in an artificial neural network. [19] Incoming input signals are weighted, then summarized through a transfer function. The resulting net input, as well as the bias (or threshold), are combined through an activation function.

Signals flow through the network over directed connections between the nodes. Every connection is associated with a *weight* value. The weights in the network can be changed; this allows the network to learn a task gradually.

Usually, when a neural network model is created, the number of nodes and the connection topology is fixed at construction time, while the weights are learned during training.

A simple form of neural network is the *feed-forward* neural network, in which signals flow only in one direction from an input layer to one or more hidden layers and then finally to an output layer.

Training a network

A neural network can be used for prediction. For this purpose, a neural network must be created with each input node corresponding to a feature. In case of categorical features, their categories can be encoded as integers; alternatively, it is possible to encode one categorical feature as a series of dummy variables

having the value 0 or 1. [3: 5.4.2] Every output node of the network will deliver a prediction, for instance a class membership probability in the range $[0, 1]$.

An neural interpretation diagram [20] of such a classification network is given in Figure 2.9. Here, we have trained a neural network on the Iris data set [9], training it to predict a flower's species (setosa, versicolor, or virginica) from the petal length and petal width.

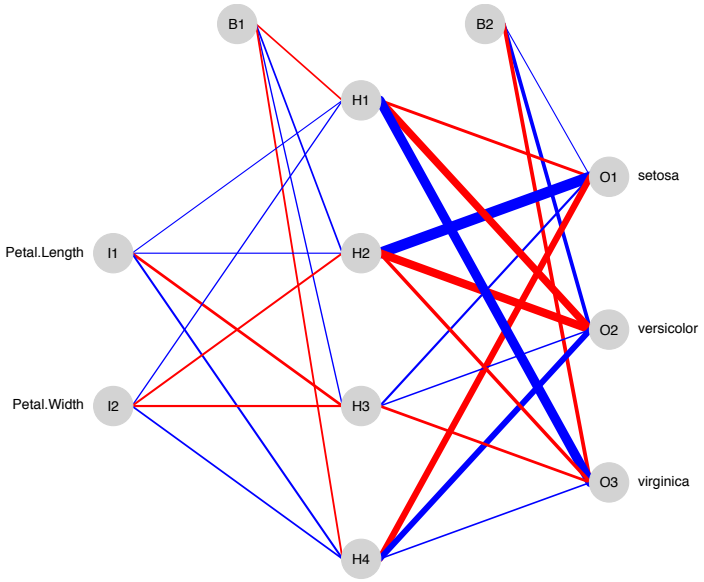


Figure 2.9: A classification network with 4 hidden nodes, trained on two features of the Iris data set to predict a class. Positive weights are shown in blue; negative weights are red. Larger absolute weights have thicker lines.

A feed-forward neural network is trained by consecutively offering training observations to its input layer, one feature corresponding to one input node. Often, input data is normalized to a range such as $[0, 1]$ or $[-1, 1]$, to conform the signal to the amplitudes of other signals in the network.

The signal travels through the neural network, modified at each node by the weights of the node's incoming connections and its activation function. During training, the output signals of the network are compared to the actual target pattern of the training example. The differences between output and actual values are minimized through numerical methods that minimize error values. Error values are communicated back from the output layer back to the input

layer, gradually adapting the connection weights to minimize the error. The errors for hidden layers are not directly related to the output; however, their weights are adapted in proportion to their assumed contribution to the error in the next layer. [21] This stepwise backward weight adaptation procedure is called *backpropagation*.

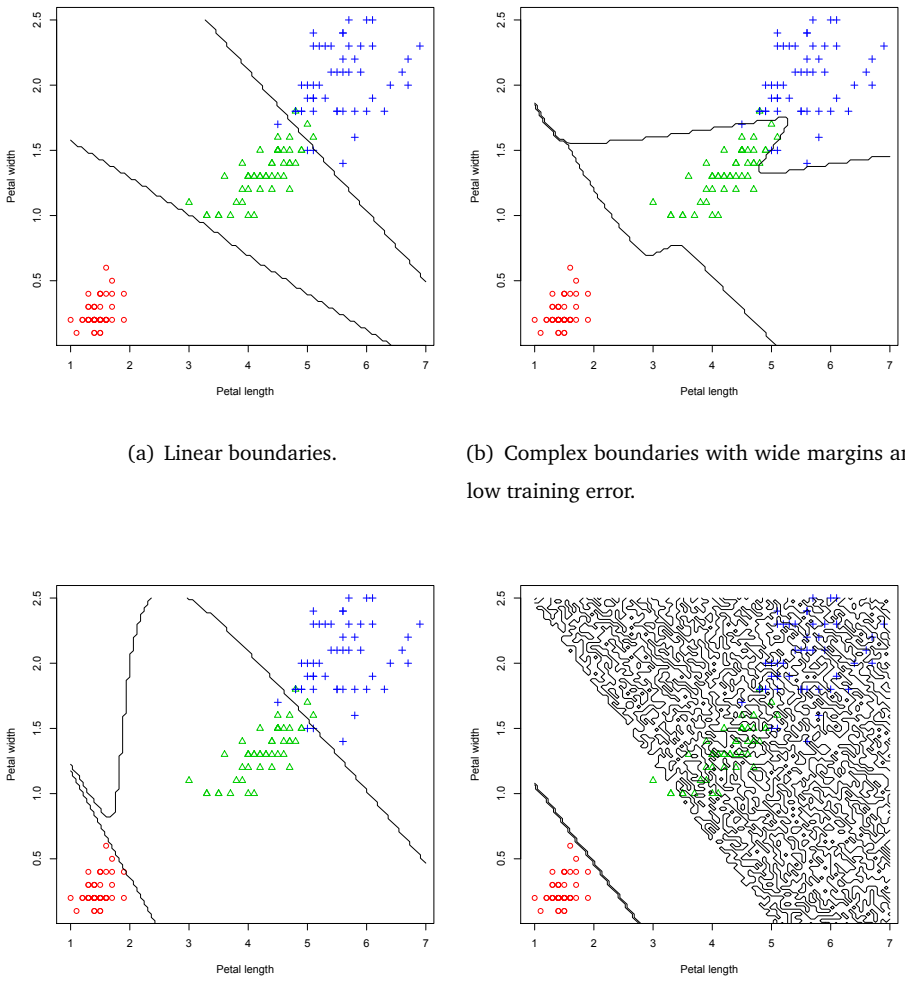
Variance of results

Learned separation boundaries and accuracies can vary dramatically from one training attempt to another, even when all parameters and the training data stays the same. This is caused by the initialization of the weights to random values at the start of training, so that the optimization problem starts from differing initial conditions. We illustrate the effect of this randomness by training a number of identical neural networks on the Iris data and plotting the resulting decision boundaries. The topology (see Figure 2.9) and hyperparameters of all neural networks are the same. In Figure 2.10, we show a selection of classifiers and their widely varying outcomes.

As the figure shows, three out of four networks have achieved reasonable separation boundaries between the classes. Cases (a) and (c) are somewhat comparable to the results attained by respectively the linear kernel SVM and the radial basis kernel SVM in Figure 2.7, although the neural network does not guarantee the wide margin that is characteristic of SVMs. Case (d) is pathological: clearly, the neural network has failed to arrive at an acceptable separation. The areas for versicolor and virginica are not properly detected and generalized. When inspecting the error values of this network during training, the iterative error minimization process was shown to be aborted because of insufficient improvements in the error function, while the error function was still very high.

This demonstrates that neural network training can easily get stuck in a local minimum, with a dramatic outcome. A possible solution is to train many neural networks and combine their results through averaging or a meta-classifier (which we will introduce in section 2.5.5).

During prediction, the neural network simply processes the values offered to its input layer by consecutively applying its weights and activation functions for every layer. In this way, it works like a function approximator. A useful property



(a) Linear boundaries.

(b) Complex boundaries with wide margins and low training error.

(c) Complex boundaries with small margins.

(d) A pathological outcome.

Figure 2.10: Prediction boundary plots for four neural networks with identical topologies trained on the same input data. Training observations are plotted as points (red circles: setosa, green triangles: versicolor; blue crosses: virginica). Neural network decision boundaries vary heavily between training attempts. Neural network (d) has failed to correctly learn the prediction task.

of neural networks is that they can be trained 'on-line' even after initial training, by offering additional training examples and performing weight adaptation.

Hyperparameters

A neural network has many selectable hyperparameters which influence its training and operation. A *weight decay* parameter introduces a penalty for higher weights, which causes weights to fall down to smaller values during training. Large weights tend to produce rough output functions. [22] Smooth and gradually responding outputs are preferred, as the resulting soft decision boundaries prevent overfitting; however, too much smoothing will increase the test error rate. The decay parameter can be tuned to determine the optimal value.

Some other parameters are the choice of transfer function, activation function, the number of hidden layers (affecting the complexity of possible patterns learned), the size (number of nodes) in each hidden layer, the initial weights (usually set to random values), and a *learning rate* which determines how quickly weights are adapted to minimize error values. [3: 5.4.2] Many modern implementations of neural networks use optimized numerical methods which do not require setting a learning rate.

The results of the neural network model will be presented in section 3.4.3.

2.5.5 Meta-classification

Combining classifiers

Using the methods described in the previous section, we can create diverse prediction models. It is interesting to investigate if we can combine these models in order to build upon the relative strengths of each model.

There are many possible strategies for combining prediction models. A simple approach could be to simply collect all the individual model predictions, and take a majority vote — or in the case of regression or class probability estimation, take the mean of the individual model outputs. However, with this strategy, a badly performing model will have a negative influence on the end result. Majority voting therefore generally results in a *worse* accuracy when compared to selecting

the best of the original classifiers through cross-validation. [23]

A solution for this problem is *stacked generalization* or *stacking*. In stacking, a new classifier is constructed which learns from the output of base models. The new classifier is called the *meta-classifier* or *level-1 generalizer*, while the original classifiers are called *base classifiers* or *level-0 generalizers*. The level-1 generalizer is said to deduce (infer) the biases of the level-0 classifiers with respect to a training set. [24] The resulting stack can make a weighted combination of the products of base classifiers, and in theory is able to generate better predictions than the best singular base classifier.

In stacking, first the level-0 classifiers are trained and tuned as described in the earlier sections. For an ensemble to improve predictions, base classifiers should provide variation. In our example, we train all the base classifiers on the same training data, but provide variation through using different model types. In other cases, variation might be induced by training base classifiers on different feature subspaces, or even from different data sources.

The level-1 data set

The level-0 classifiers are used to generate class predictions for some training set. This leads to a new *level-1* data set, which in the most simple form contains just the predictions from the base classifiers, as well as the actual class. A simple level-1 data set in our case would look like this:

sessionId	$\hat{c}_{rf}$	$\hat{c}_{svm}$	$\hat{c}_{nn}$	c
1	no	no	yes	yes
2	yes	no	no	yes
3	yes	yes	yes	yes
4	no	no	no	no

Table 2.2: Partial example of a level-1 data set. The level-1 data set contains predicted classes $\hat{c}$ that were predicted on a training set by level-0 Random Forest, SVM and neural network level-0 classifiers, as well as the actual classes c of the training set.

As a training set for this procedure, we can use the same training set used to train the level-0 classifiers. However, if we do so, we must take precautions.

In classifier error estimation (see section 2.5.1), we discussed the importance of separating the training set and test set for a classifier. If we test a classifier by predicting on its own training set, we will not get an unbiased estimate of its accuracy. The same problem holds when creating our level-1 data set: we may never use a level-0 classifier to predict on its own training set. If we do, its predictions become overconfident and will not be representative of its performance on new data. In such a situation, the level-1 classifier cannot identify its strong and weak areas.

When reusing the level-0 training set, we therefore need to build the level-1 data set by having level-0 classifiers only predict on untrained 'out-of-sample' observations. This can be done using a cross-validation procedure. [23, 25] With 10-fold cross-validation, the following steps are repeated ten times. (1) Select a new subset containing 10% of the training set. (2) Train new Random Forest, SVM, and neural network classifiers on the complement (90%) of the subset, so that the 10% subset is unseen by the classifiers. (3) Generate predictions on the held-out 10% subset from the classifiers just trained on its complement, and store these predictions in the level-1 data set. After this cross-validation loop has repeated 10 times, the level-1 data set is complete.

The level-1 data set is then used as a training set for a level-1 generalizer. The level-1 machine learning problem is therefore the prediction of actual classes from level-0 class predictions.

With the level-1 generalizer being trained and tuned, the stack is complete. Predictions can be generated from the stack by first feeding an observation to level-0 classifiers trained on the full training set, then feeding the level-0 predictions to the level-1 generalizer as input data.

Stacking with multi-response regression

There are some choices that affect the performance of a stacking classifier: first, which attributes to use for the level-1 data set; second, which machine learning algorithm to use for the level-1 generalizer.

To give an easy introduction, we introduced the concept of a level-1 data set that only contains class predictions. Ting and Witten [25] suggest that a better level-1 classifier can be created by using not the class predictions from the

level-0 classifiers, but their *predicted class membership probabilities* in the form of a real number in $[0, 1]$. This should provide the level-1 classifier a measure of how 'confident' the level-0 classifiers are in their prediction of each class. Using such a scheme, the level-1 data set in our case would look like this:

sessionId	$\hat{p}_{rf}(yes)$	$\hat{p}_{svm}(yes)$	$\hat{p}_{nn}(yes)$	$c = yes$
1	0.386	0.352	0.714	1
2	0.622	0.374	0.296	1
3	0.752	0.907	0.742	1
4	0.000	0.006	0.089	0

Table 2.3: Partial example of a level-1 data set with predicted 'yes' class membership probabilities from level-0 classifiers, and a class agreement variable which is 1 if the actual class is 'yes', 0 otherwise.

As an algorithm for the level-1 generalizer, it has been suggested [25] that *multi-response linear regression* (MLR) performs well. In MLR, a classification problem with real-valued attributes is transformed into a multi-response regression problem, i.e. a separate regression problem for each class. We fit the following linear model for each class c_i:

$$\hat{p}_{stack}(c_i) \;=\; \beta_0 + \beta_1 \hat{p}_{rf}(c_i) + \beta_2 \hat{p}_{svm}(c_i) + \beta_3 \hat{p}_{nn}(c_i) + \epsilon \qquad (2.3)$$

In this model, for a single class c_i, the probability or 'confidence' outputs from all base classifiers are weighted to arrive at a *pooled probability* measure $\hat{p}_{stack}(c_i)$. The level-1 data set is used to estimate the values of all β_j, with the dummy variable $c = c_i$ functioning as the response variable as seen in Table 2.3. The resulting model is easy to interpret manually; β_j for $j > 0$ represents the contribution of base classifier j to the correct classification of class c_i.

For a multi-class problem, the multi-response regression allows the level-1 prediction process to rely for every distinct class on the level-0 classifiers which are estimated to be most accurate in predicting that particular class. For our binary classification problem, the level-0 class predictions for 'yes' and 'no' are each other's inverse, so both their fitted gradients will also be the inverse of the other. Therefore, we need to fit only a single regression model, e.g. for the 'yes' class. Nevertheless, we will fit both models in order to check this assumption

and verify calculations; we expect the following to hold:

$$\hat{p}_{stack}(yes) + \hat{p}_{stack}(no) = 1 \tag{2.4}$$

The stack can generate a prediction by first getting base classifier predictions for an observation. These probabilities are then fed to the linear models. This leads to a stack-predicted 'pooled probability' value for every class (note that the predicted attribute in MLR need not be restricted to the $[0, 1]$ range). Prediction of an observation's class is done by picking the class c_i that receives the highest value for $\hat{p}_{stack}(c_i)$.

We could also attempt fitting a more complex level-1 model. Wolpert, introducing stacked generalization, has argued that 'relatively global, smooth' level-1 models should perform well. [24] If the level-1 model is not sufficiently smooth, it tends to overfit. [26] A support vector machine-based regression model, as used by Abbasi et al in fraud detection [27], may function well for this purpose. Therefore, after fitting a linear model based on ordinary least squares, we will also fit a model based on support vector regression.

Another possible candidate would be a multinomial logistic model. [24] However, this model seems to be rarely used in the stacking literature, and because of time constraints and data transformation requirements, it was not pursued.

The results from the meta-classifier approach will be discussed in detail in section 3.5.

2.6 Postprocessing

The goal of the DMC competition is to create a list of predictions on the test set which minimize the supplied error function defined in Equation 2.1.

We can supply predictions as floating point numbers in the range $[0, 1]$. This allows us to submit a list of predicted order probabilities which contain a measure of 'confidence' in the individual prediction results. If our classifier would have completely no preference for either 'yes' or 'no', we could submit the predicted value 0.5, for example.

On the other hand, we might just provide binary classifications (only values

0 or 1), by transforming predicted probabilities as:

$$\begin{aligned} \hat{p}(yes) < 0.5 \quad &\rightarrow \quad \hat{p}_t(yes) = 0 \\ \hat{p}(yes) \geq 0.5 \quad &\rightarrow \quad \hat{p}_t(yes) = 1 \end{aligned} \tag{2.5}$$

There could be other interesting transformation functions, such as functions that 'saturate' the predicted probability towards extremes, for instance:

$$\begin{aligned} \hat{p}(yes) < 0.25 \quad &\rightarrow \quad \hat{p}_t(yes) = 0 \\ 0.25 \leq \hat{p}(yes) < 0.5 \quad &\rightarrow \quad \hat{p}_t(yes) = \hat{p}(yes) \\ \hat{p}(yes) \geq 0.75 \quad &\rightarrow \quad \hat{p}_t(yes) = 1 \end{aligned} \tag{2.6}$$

We will simulate the effect of the most interesting transformations on the predicted outputs, producing DMC error value estimates. The transformation that produces the lowest estimated DMC error rates will be selected through cross-validation as described in section 2.5.1.

The postprocessing results will be presented in section 3.6.

Chapter 3

Results

This chapter lists the results from our analysis. The results are outlined chronologically according to the steps in the analysis process as outlined in Figure 2.1.

3.1 Software

Before reviewing the results of the classifiers, we will first discuss the software used to create them. We chose to use the statistical programming language R [28]. This open-source program is freely available, is widely used in statistics and data mining circles, and has many packages available that implement common machine learning algorithms. It allows — or forces — the user to develop an analysis pipeline as sequential program code, instead of performing actions in a graphical user interface. This orientation on code has the advantage that third parties can easily reproduce research results and re-use methods on new data sets, just by running the relevant R code. Reproducibility of analyses and peer review of authors' programming methods are important current issues in statistics and science publishing in general. [29]

Since R is extensible, anyone can contribute packages implementing custom functionality, such as a modern machine learning algorithm. For the user, this is a strength as well as a problem; for instance, it has led to many authors creating packages that provide similar functionality — often using different syntax and with subtle differences — with some of these packages suffering from sparse documentation, low code quality, or a lack of validation and testing that is inherent to commercial software. Ultimately this problem can be dealt with by defensive programming, testing algorithms on small examples, and checking

intermediate results rigorously.

During experiments, it was quickly found that computation performance of many common algorithms was problematic. For instance, a single SVM training procedure took a day to complete, even though the algorithm was run on a modern 6-processor 3.2GHz Core i7 system with Windows 8. Unfortunately, it turned out that most of the canonical R packages, such as the attempted SVM package e1071 [15], use only one processor. For historical and technical reasons, most R functions are not parallelized, as the R language and many of its packages were initially designed in a time where large multi-processor systems were rare.

Cross-validation and model tuning require training a large number of classifier models, so performance was a serious roadblock. Therefore, much effort was spent on experimenting with R packages that can make use of multi-core systems through parallelization. To enable this, the analysis had to be performed on a Linux system and later a Mac OS X system, since these operating systems offer standardized interfaces for parallelism which are not available under Windows.

The R package SPRINT [30] appeared to have promise; it implements parallelized versions of various algorithms, such as SVM and Random Forest. However, it turned out to cause errors on large training sets, which we were unable to resolve even after communication with the authors. This unfortunately wasted a lot of effort and time.

Ultimately, a solution with reasonable performance was found by the realization that *within* a model tuning procedure, the many training procedures are not dependent on each other. Only at the end of cross-validation, the results of each individual model training procedure need to be collected and compared. Therefore, a speedup can also be realized by training the various models in parallel on different processors.

An R package that provides parallel tuning is caret [5], a utility package for classification and regression models. This package can coordinate large tuning jobs by making use of basic parallelism functions. These functions are not part of standard R, but are provided by the foreach package [31]. There are various 'back-ends' for the foreach package, some aimed at large computing clusters, some aimed at smaller systems; we used the doMC package [32] which spreads workloads across multiple processors of a single multi-core system. This allowed

us to tune complex models within a reasonable time frame.

Another significant advantage of caret is its common interface to a multitude of classification packages. These packages are vetted carefully, and any differences in their usage are abstracted away by a common syntax. This decreases the possibility of human error and allows the user to try various classifiers on a single data set with small effort. Nevertheless, tuning and cross-validation loops on a data set of our size are still rather long, varying from multiple hours to days, so we were not able to include additional classifier models in our search.

3.2 Feature extraction

Preprocessing of data is an important step in the DMC problem. As detailed in section 1.2, a data set is provided for individual transactions only; however, we must create predictions for a visitor *session*, potentially consisting of many transactions. Therefore we must extract aggregated session features from the transaction data. For instance, based on exploration of the data, we can extract features such as:

- total number of transactions in the session;
- average price of products viewed;
- maximum number of products put in the shopping basket.

We can use various aggregation operators, such as *count, sum, average, max, min, range, standard deviation,* and *gradient*. These may reveal sub-patterns present in the various transactions constituting the session. For instance, a low *range* of viewed prices might reveal that a visitor is specifically visiting related product pages; a positive *gradient* in the web-shop order steps could hint at an increasingly confident visitor; a high *count* of transactions shows that a visitor spent a long time on the website. Features such as these might be informative to determine if a visitor is likely to order a product.

We have attempted all reasonable combinations of transaction variables and aggregation operators. Some of the aggregation functions may not be semantically meaningful in the problem domain (for example taking the minimum of a series of *max viewed price* measurements). However, we have tried not to

exclude features too quickly. The stated descriptions of the transaction variables are not always very clear (see section 2.3), so we might miss some information by being too restrictive in feature extraction.

The number of features generated must of course remain tractable in the context of machine learning methods. The model fits might later demonstrate which features are informative and which can be removed from further analysis. If computation time turns out to be prohibitive, we can choose a limited set of features to retain.

Some variables in the transaction data are static throughout a session, and these can be directly used in the session data. The variable *customerID* is omitted before training, as it is an identifier and not a proper predictor. Using an identifier variable to train a classifier might lead to overconfidence in accuracy estimations, as future observations may be measured on different customers and the model may then turn out to be ill-optimized for those observations.

The full set of extracted features is given in Table 3.1.

sessionNo	countTransact	startHour	startWeekday
sumDuration	minCCount	maxCCount	avgCCount
minCMinPrice	avgCMinPrice	maxCMinPrice	sdCMinPrice
gradCMinPrice	rangeCMinPrice	minCMaxPrice	avgCMaxPrice
maxCMaxPrice	sdCMaxPrice	gradCMaxPrice	rangeCMaxPrice
minCSumPrice	avgCSumPrice	maxCSumPrice	sdCSumPrice
gradCSumPrice	rangeCSumPrice	minBCount	avgBCount
maxBCount	minBMinPrice	avgBMinPrice	maxBMinPrice
sdBMinPrice	gradBMinPrice	rangeBMinPrice	minBMaxPrice
avgBMaxPrice	maxBMaxPrice	sdBMaxPrice	gradBMaxPrice
rangeBMaxPrice	minBSumPrice	avgBSumPrice	maxBSumPrice
sdBSumPrice	gradBSumPrice	rangeBSumPrice	minBStep
avgBStep	maxBStep	minOnlineStatus	maxOnlineStatus
minAvailability	avgAvailability	maxAvailability	maxVal
customerScore	accountLifetime	payments	age
address	lastOrder		

Table 3.1: Extracted features for a session. The features are aggregated from transaction data.

Sometimes, data analysts apply multivariate functions to existing features, for instance generating products or ratios. This is especially useful if these derived

features have a concrete meaning or application in the problem domain. We have chosen not to generate such derived features, as the added dimensions would likely increase computation time significantly, while we will choose data mining algorithms which themselves should be able to deal with multivariate patterns efficiently.

The reading, parsing, feature extraction and output writing steps were programmed using version 5.4.13 of the PHP language [33]. While this language is not used often in data analysis, it is well-suited for creating short data processing scripts. The feature extractor has the following components:

- **Converter:** Main program that calls the consecutive steps to convert transaction files in DMC format to feature data sets for training and testing.
- **FileReader:** The DMC file is read and parsed to rows of transaction data. Categorical string values *availability* and *order* are mapped to numerical values. The DMC value '?' is mapped to the R value NA (Not Available).
- **SessionSorter:** Each transaction is associated with a single session on its *sessionId* variable. The transactions are now chronologically sorted per session.
- **FeatureExtraction:** For each session, data from its transactions is aggregated, and static session data is copied.
- **FileWriter:** The session features are written to a training CSV file and a test CSV file.

There are 429,014 transactions in the training data file `transact_train.txt` and 45,069 transactions in the test data file `transact_class.txt` [2]. As multiple transactions are associated with one session, the transaction data represents 50,000 training sessions associated with a known outcome, and 5,111 test sessions to be used as new input data for predictions.

3.3 Missing values

Exploration showed that many of the available variables in the transaction data have large occurrences of missing values. This translates similarly to missing

values in the extracted session features. In section 2.4, we discussed various strategies for dealing with missing values in the data set. We execute and evaluate these strategies.

3.3.1 Excluding missing values

As discussed in section 2.4.1, excluding missing values requires some conditions to hold: the frequency of missing values should not be too high, and the occurrence of missing values should be independent from the outcome. We test for these conditions by counting the numbers of complete observations, and breaking down the frequencies of successful orders for all observations and only the complete observations respectively.

	n	p_{order}	$p_{no-order}$
Complete training sessions	22,237	0.25	0.75
All training sessions	50,000	0.54	0.46

Table 3.2: The training set contains relatively few fully complete observations. There appears to be a strong relationship between missing values and outcome.

From Table 3.2, it follows that simply disregarding observations with missing values is not a viable strategy for this data set. More than half of the observations contain at least one missing value, and the outcome appears strongly associated with the presence of missing values. For the complete observations, 25% of sessions end in an order. For all observations, this is 54%. Therefore, to maintain a representative training set, we must use the full set of sessions, and choose an imputation method.

3.3.2 Imputation methods

As we cannot simply exclude incomplete observations, we must explore proper imputation methods.

Mean imputation (explained in section 2.4.2) is available in the R imputation package as `meanImpute`.

Predictive imputation (see section 2.4.3) is available through various functions in R. We benchmark a predictive imputation method based on *boosted trees*,

available as `gbmImpute` from the imputation package [34]. This function trains regression trees for all variables that have missing values. A regression tree is a decision tree that has a continuous response variable. Boosting is used to improve accuracy of the prediction. A short description of boosting is given in section 2.5.2.

Unique-value imputation (described in section 2.4.4) is not available as an R function. Therefore we created this procedure in custom R code. The `uvImpute` function implements the following algorithm: loop through all columns of the data set, for every column doing the next steps: start with –1 as 'unique value'; check if this value exists in the column, and if so, decrease the unique value to –2, –3, et cetera, until the value does not exist in the column; replace all missing values by the current unique value. Note that our algorithm is slightly different from the approach suggested by Sariyar et al. [35] who have imputed a separate unique value for each observation; their task however was duplicate detection, in which the expression of *inequality* between imputed observations is important. We do not expect this to be necessary for our classifier, but we will adjust the algorithm if empirical evaluation shows low accuracy.

3.3.3 Evaluation

For an imputation method to be useful, it should retain the information in the original data set and not obstruct the training process. We evaluate the usefulness of the various imputation methods by using the resulting data sets to train a classifier, followed by accuracy estimation of the classifier.

To evaluate classification accuracy for the imputation approaches described, we have fit Random Forest classifiers using imputed data sets as training sets. We have fit Random Forest classifiers with default settings, using the `randomForest` function from the randomForest R library [11]. As defaults, the library used 500 trees and 7 variables randomly sampled at each split.

Determining the evaluation result quickly was very important, as imputation was a necessary first step and detailed model evaluation could not start until the imputation methods were ranked. We used only Random Forest for imputation selection, since this learning algorithm has few tuning parameters and offers an embedded 'out-of-bag' error estimate (see section 2.5.2). This allowed for

very quick training and evaluation without the need for long cross-validation procedures.

Time constraints forced us to make the assumption that the performance of Random Forest on the imputed training set serves as a good proxy for the performance of other classifiers to be constructed later. To check the validity of this assumption, we have later also trained and tuned support vector machine classifiers on all imputed training sets, and obtained an accuracy estimate by 10-fold cross-validation. This process took considerably longer, but it could run concurrently with the rest of the analysis.

To explore changes in data structure due to the various imputation methods, we have trained decision trees on data sets imputed by the three methods. The trees were fitted using the R tree package [8]. The results are shown in Figure 3.1. The trees do not seem to show a lot of structural differences in their most significant splits, although the unique-value tree is deeper and includes the 'age' feature. Also, the split value for 'maxBStep' is influenced somewhat, as observations with a missing value for this feature have influenced its distribution.

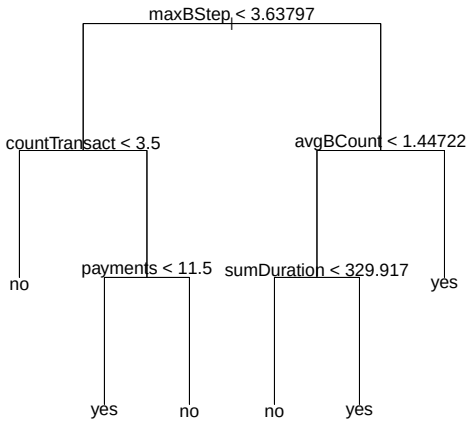
Imputation method	**Random Forest accuracy**	**SVM accuracy**
Mean imputation	0.8933	0.876
Boosted tree imputation	0.8937	0.876
Unique-value imputation	0.8937	0.869

Table 3.3: Accuracy estimates of classifiers trained on data sets imputed by various methods.

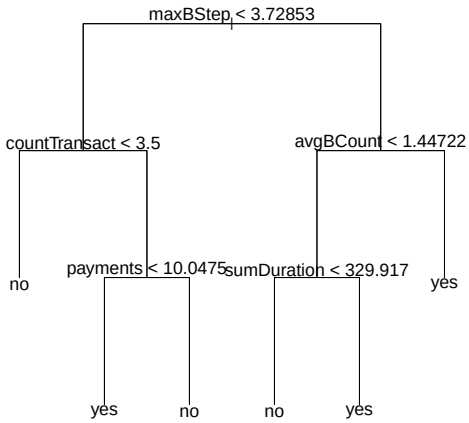
3.3.4 Selection

Table 3.3 interestingly shows that none of the imputation methods appear to have a significant advantage over the others in terms of classifier accuracy. Mean imputation appears to have a slight disadvantage for the Random Forest classifier, with both predictive and unique-value imputation performing having the best result.

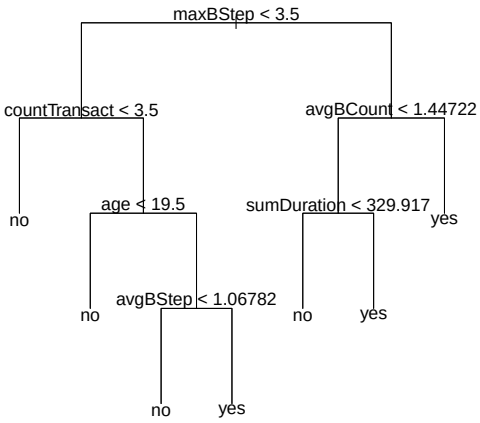
While we did not make any determinations of the variance in estimates, the Random Forest numbers already appear so similar that we selected the *unique-value imputation* approach for its speed, simplicity, and easy adaptability to new



(a) Mean imputation



(b) Boosted tree imputation



(c) Unique-value imputation

Figure 3.1: Decision trees trained on data sets imputed by various methods.

data.

The information obtained from the support vector machine classifiers later shows that unique-value imputation appears to perform worse on the SVM classifier, but only slightly. Mean imputation and predictive imputation appear to have the same accuracy. Again, any differences seem to be so small that they will not have any significant impact on the general analysis; quite possibly they are due to random chance. This did not cause us to reconsider the imputation approach.

3.4 Model training and evaluation

In section 2.5.1, we sketched the process for obtaining accuracy estimates of various prediction models by training and cross-validation. Model training was performed using the `train` function of the caret package [5, 36] in R. This package provides a standardized interface to create various prediction models, and offers integrated model tuning by cross-validation. Cross-validation was performed using the cvTools package [37] and custom code, e.g. for training the meta-classifier on cross-validation folds and for implementing the specific DMC accuracy function in Equation 2.2 as an optimization target. We perform DMC error estimation using a custom function passed to the `trainControl` parameter in the caret package.

We perform no feature selection before model training. The number of features (see Table 3.1) falls easily within the tractable range for prediction modeling. Even though the required computing time for the cross-validated training procedures was shown to be rather long (often 8 to 24 hours per model type), it was acceptable in the context of this project. Feature selection itself also complicates cross-validation, since selection of the features itself should not be done globally but be part of the separate cross-validation steps; otherwise, it can introduce a bias which produces overly optimistic test error estimates. An extreme example of this is given by Hastie et al. [7: 7.10.2]

3.4.1 Random Forest

Tuning

In section 2.5.2, we discussed the use of Random Forest and its hyperparameter m_{try} which we need to optimize for best performance. We use the caret `train` function with the randomForest package [5,11] to build several models and train the final model automatically after determining the optimal value for m_{try}.

For tuning, caret make use of 10-fold cross validation, which provides a good compromise between computational effort (around 4 hours on a 6-processor 3.2GHz Core i7 system) and reliability of the result (each model is trained on 90% of the training set). The results are presented in Figure 3.2 and tabulated in Table A.1.

We have attempted 8 values of m_{try} during tuning. As becomes clear from the accuracy curve, the optimal cross-validated error estimate is attained at $m_{try} = 28$. With this choice of m_{try}, the estimated binary classification accuracy is 0.898 and the estimated DMC accuracy (see Equation 2.2) is 0.796. Note that the accuracy curve is almost flat around the optimum, demonstrating that Random Forest appears not very sensitive with regard to the choice of m_{try} on this training data. Standard deviations of both accuracy estimates are very low, giving high confidence in the estimates.

Out-of-bag measures

As discussed, we will not perform feature selection in our analysis. However, for purposes of model inspection, we will look at the embedded variable importance measures returned by the Random Forest model (see section 2.5.2 for background information). A plot of these importance measures is provided in Figure 3.3. The R randomForest library implements multiple measures; we show the mean decrease in accuracy after permuting the feature.

We find that some features have relatively high significance, such as maxBStep (the maximum reached stage in the order process), avgBStep (the average stage across transactions; similar to maxBStep), countTransact (the number of transactions), maxAvailability (the maximum value of product availability indicators – if viewed products are more likely to be in stock the probability of

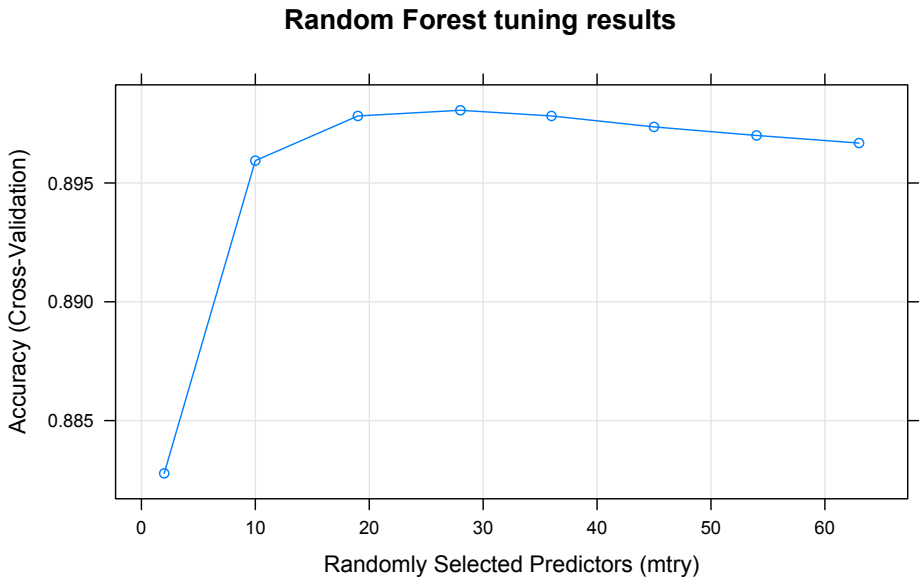


Figure 3.2: Results of tuning Random Forest to optimize accuracy using cross-validation. Although all choices of m_{try} demonstrate good estimated accuracy, the optimal estimated accuracy is attained at $m_{try} = 28$.

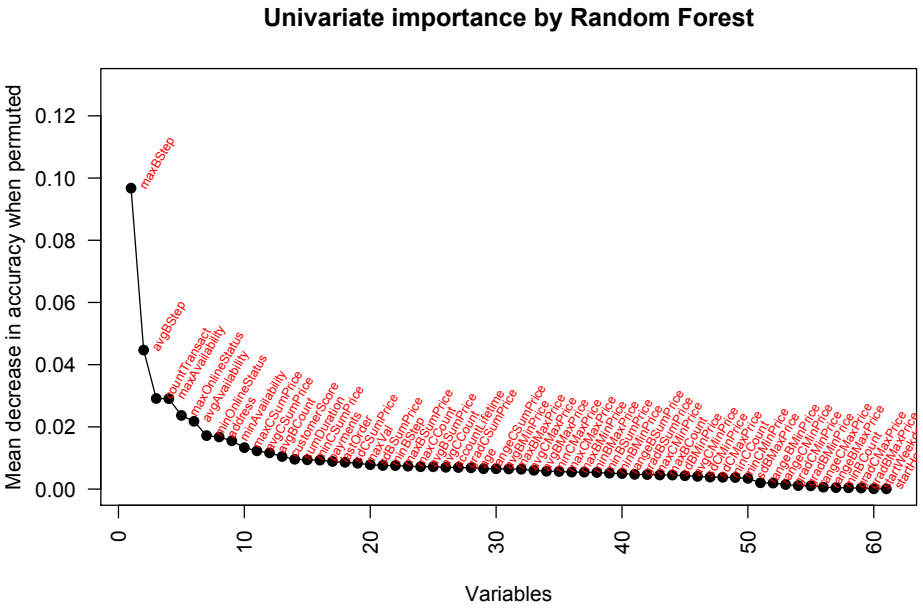


Figure 3.3: Random Forest variable importance estimates, sorted from most important feature to least important feature.

ordering is higher) and maxOnlineStatus (indicating whether the customer stays online).

Interestingly the importance curve only levels off very slowly afterwards. It appears that most other features are not highly predictive, but still provide limited information, although it remains unclear at what part variables start to become just noise.

There are no variables with a negative importance, which would mean that the Random Forest functions better without including that variable. Since we are interested in even small accuracy advantages over competitors, we have retained all variables for analysis and have not retrained the model with a subset of features.

Accuracy details

Finally, we explore the estimated accuracy of predictions, and look at how n_{tree}, the number of trees, influences this accuracy. The Random Forest algorithm automatically computes 'out-of-bag' error estimates for the binary classification case. A plot of the overall out-of-bag error estimates, error estimates for the 'order' class, and error estimates for the 'no order' class, is presented in Figure 3.4. The curve shows that $n_{tree} \simeq 100$ already produces good accuracy, but increasing the number of trees still yields additional accuracy up to $n_{tree} \simeq 300$.

As the confusion matrix in Table 3.4 shows, there is a slight difference in out-of-bag error rates for the two classes, with 'no order' observations more likely to be misclassified at a rate of 11.56% versus 8.48% respectively.

actual class	observations	no predicted	yes predicted	out-of-bag error
no	26822	23722 (88.44%)	3100 (11.56%)	11.56%
yes	23178	1965 (8.48%)	21213 (91.52%)	8.48%

Table 3.4: Confusion matrix for out-of-bag predictions.

The optimal overall out-of-bag error estimate is 10.13%, which results in an expected test error of 1–0.1013 = 0.8987. This number is very similar to the cross-validated error estimate 0.898 determined during cross-validation tuning.

This demonstrates that the Random Forest out-of-bag error rate is a useful proxy for the cross-validated error estimate, although of course when only

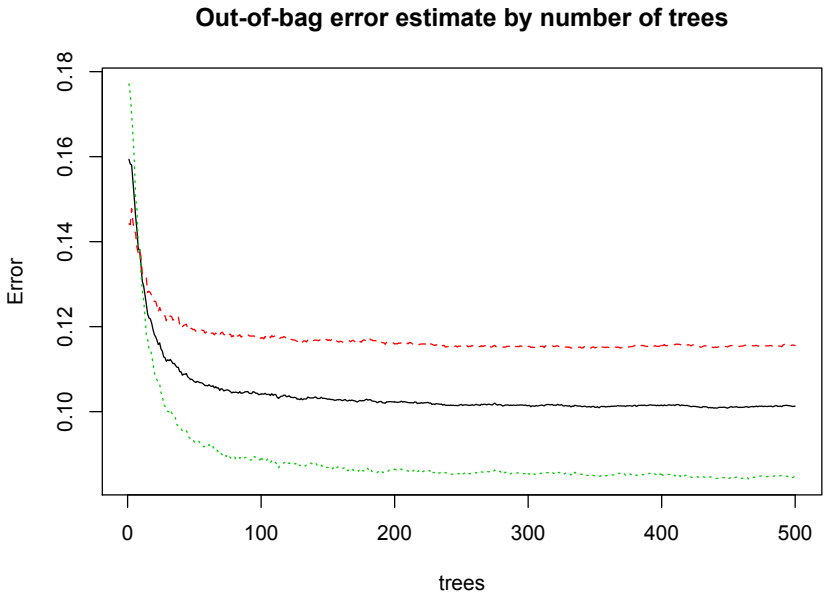


Figure 3.4: Random Forest out-of-bag error estimates plotted against number of trees. Black: overall error. Red: 'no' error. Green: 'yes' error.

depending on the out-of-bag error we do not receive a standard deviation for the estimate. Further, by using cross-validation on all models, we ensure that error estimates are comparable across multiple prediction models.

3.4.2 Support vector machine

Tuning

In section 2.5.3, we sketched the support vector machine approach and the various choices to be made when applying the method. One of these choices is the kernel function. Ideally, one would try various kernels on a data set. Initial exploration showed that training an SVM was computationally very expensive, most likely due to the large number of training observations. Because of time constraints, we work only with one kernel. We select the radial basis function (RBF) kernel, which should give good results on a non-sparse data set such as ours. [14]

As an SVM implementation, we have used the caret `train` [5] method with

the `ksvm` classifier function from the kernlab package [38]. The RBF kernel was pre-selected. Tuning was aimed at finding optimal values for cost parameter C and the RBF kernel's width parameter γ, which were discussed in section 2.5.3. Both hyperparameters influence the complexity of the decision surface and thus strike a balance between overfitting and underfitting.

In the kernlab package, a *Gaussian* kernel is used as an RBF kernel. For the Gaussian kernel, a width parameter σ is optimized instead of the inverse-width γ; these are related by $\gamma = 1/2\sigma^2$, so tuning the other formulation provides equivalent results. In our tuning tables and figures, we will use the σ parameter.

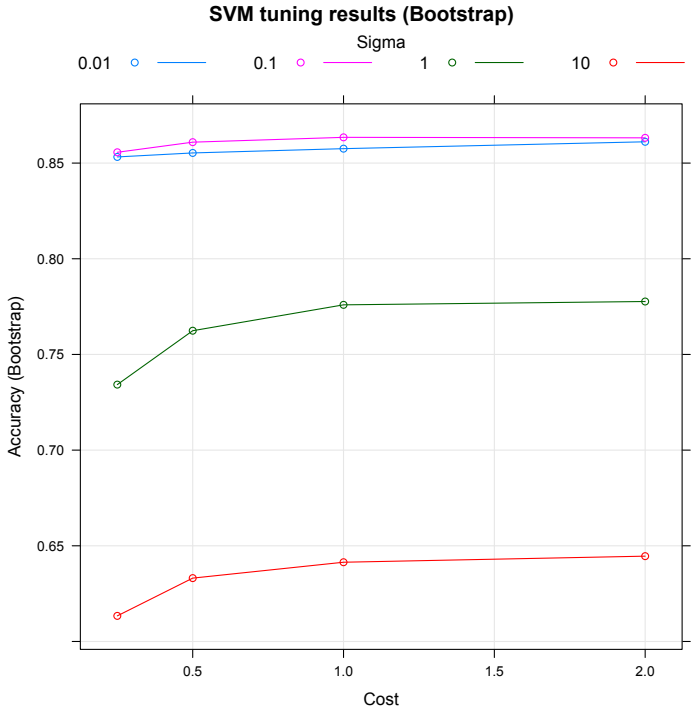


Figure 3.5: Results of first SVM tuning attempt using 25-repeated bootstrap.

Grid tuning

We performed an initial SVM tuning. Tuning was performed on a 4×4 grid with $0.25 \leq C \leq 2$ and $0.01 \leq \sigma \leq 10$, with four values attempted for both parameters. The values attemped are listed in Table A.2. Hyperparameters selected during this process were $C = 1$ and $\sigma = 0.1$. The results are plotted in

Figure 3.5 and tabulated in Table A.2. The estimated accuracy when making binary class predictions is 0.869; the estimated DMC accuracy based on probability predictions (see Equation 2.2) is 0.738. We note that the error curve is largely flat around this value of C.

Due to an unfortunate programming error in calling the `trainControl` function, caret's default setting of *25-times bootstrap resampling* as an error estimation approach was used instead of cross-validation. (See section 2.5.1 for an explanation of this method.)

Due to the multiple-day computation time involved with training 4×4 SVMs, we chose not to repeat the full tuning procedure using cross-validation. This means that test error estimates from this bootstrapped procedure might not be directly comparable with the other (cross-validated) estimates. However, the ranking of accuracies from hyperparameter selection *within* the tuning procedure should be representative. [7: 7.12]

Sigma-optimized tuning

The 4×4 tuning grid used in the first tuning run was still rather coarse out of necessity due to the long calculations. Therefore, it might not have hit the exact optimal value for σ. The kernlab library [38] provides a `sigest` (sigma estimation) function which estimates a good value for the Gaussian kernel's σ (width) parameter for a certain data set.

We recall from section 2.5.3 that the ideal kernel should be wide for a sparse data set, and narrow for a dense data set. The `sigest` function estimates the density of the training data, and determines a good σ accordingly. [38]

The resulting `sigest` value on the training set was $\sigma = 0.0371$. This value for σ was kept constant, while caret's default values for the cost parameter $C \in \{0.25, 0.5, 1\}$ were attempted. This tuning procedure was performed using 10-fold cross-validation, as for the other model types.

In this run, the best accuracy estimate was attained at $C = 1$. As in the earlier tuning run, differences between estimates are very small with respect to changes in C. The variance of the estimates is almost zero. The results are tabulated in Table A.3.

When we compare Table A.2 and Table A.3, bootstrapping and cross-validation

methods produce error estimates which are remarkably similar. The cross-validated tuning attempt gives $C = 1$ and $\sigma = 0.0371$, with a cross-validated test accuracy estimate of 0.869 for binary predictions or 0.738 for probability predictions.

Tuning of the second model was much faster, because of the need for fewer attempts as the value of σ was held constant. There appears little profit in fitting a larger range for C. Therefore, we select the SVM with $C = 1$ and $\sigma = 0.0371$ for the remainder of the work.

3.4.3 Neural network

Tuning

We discussed the properties of neural networks in section 2.5.4. The R nnet library simulates feed-forward neural networks with a single hidden layer. [39] We use the caret `train` function [5] to tune neural networks over various hyper-parameter settings and estimate their accuracies using 10-fold cross-validation.

We tune to find optimal values for *size* (number of nodes in the hidden layer, affecting complexity of patterns learned) and *decay* (weight decay parameter, which influences smoothness of output) on the data set, as detailed in section 2.5.4. We expect accuracy to rise in response to a certain size increase, with the weight decay choice leading to an optimum between underfitting and overfitting. The cross-validated accuracies are plotted in Figure 3.6; they are tabulated with their standard deviations in Table A.4.

The tuning result is interesting for many reasons. First, the accuracy estimates are significantly worse than those of SVM and Random Forest models. Second, there appears to be a much larger variance in accuracy estimates compared to these models. This variance is reflected in the seemingly multimodal tuning graphs and the larger standard deviations of the cross-validated estimates. In contrast to the other models, there does not appear to be a clear optimum for any combination of parameters. Reasons for this might lie in the influence of random weight initializations at the start of training, and the presence of local minima in the numeric optimization process causing failure in some of the models trained during cross-validation.

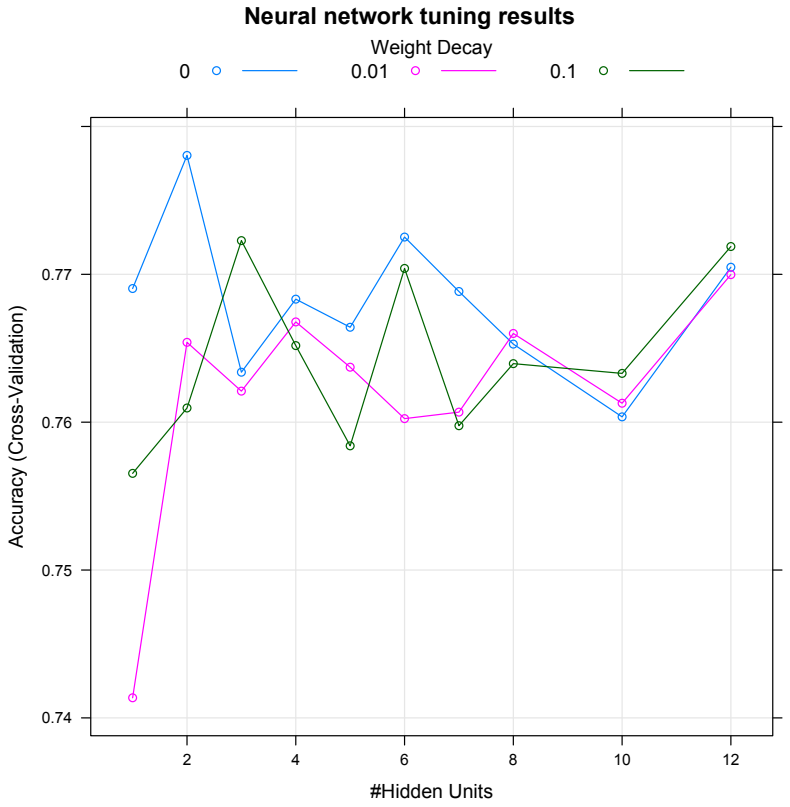


Figure 3.6: Results of neural network tuning for size (number of hidden nodes) and weight decay. Hyperparameters selected were $size = 2$ and $decay = 0$.

Another striking result is that increasing the size of the hidden layer beyond $size = 2$ does not seem to significantly improve the accuracy. Apparently, the neural network method only uses some linear combinations of features, and is not able to make use of sparse patterns in the data in order to increase accuracy.

The best accuracy appears to lie at $size = 2$ and $decay = 0$, although the difference is small. The result is remarkable: this means that the winning neural network does not make use of weight decay smoothing. With these parameters, estimated binary class prediction accuracy lies around 0.778 and DMC accuracy lies around 0.561, which is significantly lower than the accuracy of Random Forest and SVM.

Due to the large variance of the estimates, it is hard to select the best performing model; it might be that many other networks are equivalent in accuracy and any relative differences are due to chance. In this case, we select

the simple 2-node model, according to Occam's razor: when choosing between two models with the same generalization error, the simpler model must be preferred. [3: 4.4.4]

Interpretation diagram

A neural interpretation diagram [20] of the selected model is plotted in Figure 3.7. This diagram shows the nodes and connections in the neural network. As discussed, information flows from the input layer (on the left) to a hidden layer and finally to the output node. Connections having positive weights are shown in blue; connections having negative weights are colored red. Larger absolute weights produce a thicker line, meaning that the value on the left has a larger influence on the right node.

The neural interpretation diagram allows us to derive that the model has detected two latent 'factors' influencing the output. Hidden node H1 has a negative contribution to the order probability, and hidden node H2 has a positive contribution. Features weighed heavily are: countTransact, minCCount, maxCCount, gradCMaxPrice, sdCSumPrice, and address. It is interesting to note that the list of features with large weights only partly overlaps the set of most important variables according to Random Forest's embedded variable importance metric shown in Figure 3.3.

3.4.4 Model evaluation

In the preceding sections, we presented our tuned Random Forest, SVM, and neural network classifiers. Before continuing, we summarize and evaluate the estimated accuracies of these classifier models (see section 2.5.1 for details). If there are any models with extremely bad accuracy, we can remove them from further analysis. The remaining models will become part of the meta-classifier.

The cross-validated accuracies of the three models are presented in Table 3.5. The Random Forest and support vector machine methods appear to give the best results on the data set, with Random Forest having a slight accuracy edge. The neural network trails these accuracies by quite a large margin; however, its result is still better than what would be expected by chance. There is a possibility that the neural network might still provide some useful information in edge cases,

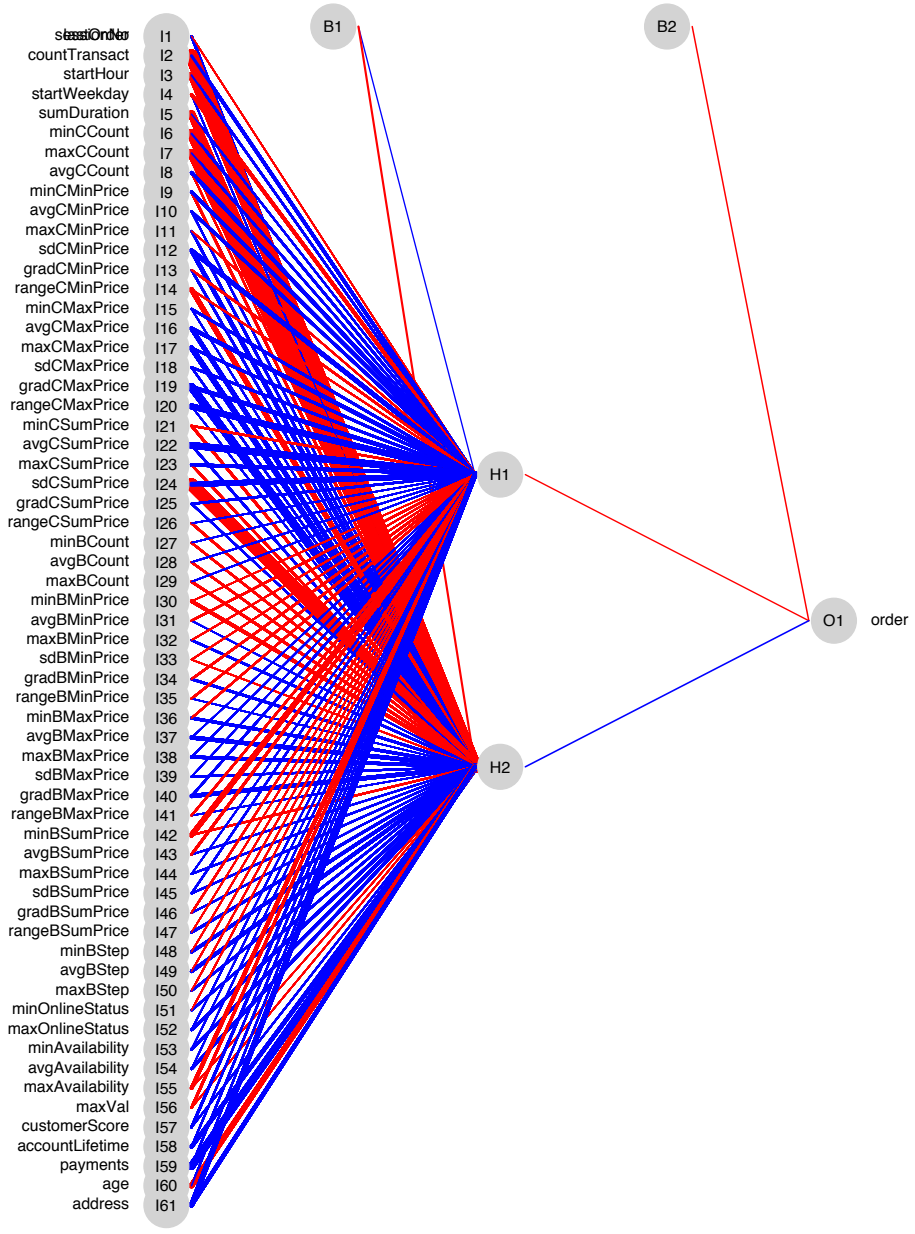


Figure 3.7: Neural interpretation diagram of the final neural network with two hidden nodes. Positive weights are blue; negative weights are red. Larger absolute weights have thicker lines.

Model	Binary accuracy	Binary SD	DMC accuracy	DMC SD	Rank
Random Forest	0.898	0.005	0.796	0.011	1
SVM	0.869	0.004	0.738	0.008	2
Neural network	0.778	0.027	0.561	0.052	3

Table 3.5: Cross-validated accuracy estimates of tuned classifiers, standard deviations of the accuracy estimates, and the estimated rankings of the classifiers.

for instance by classifying correctly some observations that were misclassified by both Random Forest and SVM classifiers. Therefore, all classifiers are retained in our meta-classifier.

Because of time constraints, the meta-classifier could not be finished in time for submitting the results to the DMC competition. Therefore, we have submitted to the competition the best single classifier selected by cross-validation, which is the Random Forest classifier.

3.5 Meta-classifier

3.5.1 Overview

As explained in section 2.5.5, the building of a stacking meta-classifier using MLR and level-1 probabilities comprises a number of steps. We have implemented the following procedure:

1. Tune base classifiers: Random Forest, SVM and neural network;
2. Use the base classifiers to generate a level-1 data set containing probability predictions for the training set, using a 10-fold cross-validation procedure to predict on out-of-sample data;
3. Fit level-1 models for the ‘yes’ and ‘no’ classes, using the actual class in training data as a response variable;
4. Generate predictions from the stacks by predicting pooled probability values for each class, and picking the class that has the highest predicted probability.

The last two steps of this procedure were followed two times: once using ordinary least squares linear regression as a level-1 generalizer, once using SVM regression instead. There is no available stacking implementation for R; there is a work-in-progress package called caretEnsemble [40] which purports to implement stacking, but unfortunately this package was not in a working state at the time of writing. Therefore the algorithms had to be implemented from scratch.

3.5.2 SVM probability correction

As discussed, we derive our level-1 data set from class membership probability estimates predicted by the base classifiers. During checking of intermediate results, the probability predictions of the base SVM classifier were somewhat unexpected. All models, except for SVM, predicted the class for an observation through the following equality:

$$\begin{aligned} \hat{p}(yes) < 0.5 \quad &\Leftrightarrow \quad \hat{c} = no \\ \hat{p}(yes) \geq 0.5 \quad &\Leftrightarrow \quad \hat{c} = yes \end{aligned} \tag{3.1}$$

In other words, the probability boundary between 'yes' and 'no' classes was at $\hat{p}(yes) = 0.5$. This property did not hold for the SVM classifier. The 10 SVM classifiers trained during the cross-validation prediction procedure instead had varying boundary values around $\hat{p}(yes) = 0.415$, the exact boundary depending on the training fold. This observation is sketched in Table 3.6.

As can be seen, for each fold the probability boundaries of 'no' and 'yes' predictions are slightly different. The separation value in this case may be due to an imbalance in classes of our training set (46% 'yes', 54% 'no').

Standard SVM do not generate probabilities; however probability models can be added to the SVM. [41] Various approaches have been suggested to calibrate SVM probabilities, for instance histogram-based techniques. [42] However, no implementation-related issues seem to be discussed in the documentation of the kernlab library [38]. Its documentation simply claims that class probabilities are supported without any discussion of calibration. We assume it uses a proper probability model, but it remains an open question if kernlab can be configured to produce fully calibrated probability predictions. Another possibility could be to try weighting of classes during SVM training to see if this imbalance disappears.

Fold	$max\ \hat{p}_{svm}(yes) \mid \hat{c}_{svm} = no$	$min\ \hat{p}_{svm}(yes) \mid \hat{c}_{svm} = yes$	$p_{boundary}$
1	0.41547	0.41773	0.41660
2	0.41704	0.41975	0.41839
3	0.41555	0.41765	0.41660
4	0.41753	0.41821	0.41787
5	0.41570	0.41692	0.41631
6	0.41394	0.41471	0.41432
7	0.41584	0.41728	0.41656
8	0.41511	0.41702	0.41606
9	0.41464	0.41879	0.41672
10	0.41470	0.41700	0.41585

Table 3.6: Ranges of observed predicted probabilities for 'no' and 'yes' classes by SVM base classifier.

Nevertheless, due to the long computation time of the meta-classifier, re-training the full stack was not a viable option.

The unbalanced probabilities might pose a problem for the level-1 generalizer. First, 'yes' cases in the training set coincide with lower SVM-predicted probabilities, possibly leading the level-1 model to conclude that the SVM classifier is less accurate in predicting the 'yes' class. Second, due to overlapping ranges between the various folds, simple combination of the folds will result in some 'yes' predictions having a *lower* associated $\hat{p}_{svm}(yes)$ than other 'no' predictions (for instance, in fold 6 there is a 'yes' prediction with $\hat{p}(yes) = 0.41471$, while in fold 4, there is a 'no' prediction with $\hat{p}(yes) = 0.41753$). For those reasons, we perform a quick correction on the $\hat{p}(yes)$ values predicted by the SVM base classifier.

Therefore, we performed a simple linear scaling operation to the $\hat{p}_{svm}(yes)$ values to shift the predicted class boundary to a corrected $\hat{p}_{corr}(yes) = 0.5$:

$$\hat{p}_{corr}(yes) = \begin{cases} \frac{0.5}{p_{boundary}}\ \hat{p}(yes) & \text{if } \hat{p}(yes) < p_{boundary}; \\ 1 - \frac{0.5}{1-p_{boundary}}(1 - \hat{p}(yes)) & \text{if } \hat{p}(yes) \geq p_{boundary}. \end{cases} \tag{3.2}$$

Here we define $p_{boundary}$ as the mean of the highest observed $\hat{p}_{svm}(yes) \mid \hat{c}_{svm} = no$ and the lowest $\hat{p}_{svm}(yes) \mid \hat{c}_{svm} = yes$ on the training fold.

After applying these corrections to the predictions in every fold, the level-1 data set can be composed from all corrected folds.

When making predictions from the stack, we must apply the same correction to the level-0 SVM classifier's predictions to then use as level-1 features.

3.5.3 Ordinary least squares linear regression

On the level-1 training set in a form described by Table 2.3, we fitted the following linear model, using the pooled order probability as a response variable and the base classifier probabilities as predictors:

$$\hat{p}_{stack}(yes) \;=\; \beta_0 + \beta_1 \hat{p}_{rf}(yes) + \beta_2 \hat{p}_{corr}(yes) + \beta_3 \hat{p}_{nn}(yes) + \epsilon \tag{3.3}$$

We fitted the model for the 'yes' class. Using the `lm` function of R's built-in stats package [28], the following coefficients were estimated:

Coefficient	Interpretation	Estimated value
β_0	intercept	—0.01165
β_1	weight of base RF	0.93749
β_2	weight of base SVM	0.08670
β_3	weight of base NN	0.00297

Table 3.7: Fitted values for level-1 linear regression model.

The results of the linear model allow for easy interpretation. [43: Ch. 7–8] The Random Forest base classifier accounts for most of the variation in the response, with the SVM base classifier acting as a weaker predictor. The neural network base classifier appears to have almost no influence on the predictions of the stack.

This finding correlates well with the cross-validated accuracy estimates of the base classifiers in section 3.4.4. The Random Forest classifier was estimated by cross-validation to have the highest accuracy, so it follows that it plays a large role in the stack. The SVM classifier trailed behind slightly in cross-validation, and indeed its contribution in the linear model largely overlaps that of the Random Forest, but it is still included in the stack. The neural network was already shown to perform a lot worse in cross-validation. It was hoped that the neural network with its vastly different algorithm could still contribute to the accuracy of the stack in a minor way, but apparently this is not the case.

The linear model above was fitted for the 'yes' class. As discussed in section 2.5.5, the original MLR algorithm specifies that a regression model must be fitted for every class, in order to pick the class with the highest pooled probability predicted by its model. Since we have a two-class problem, we expected the fitting of a model for the 'no' class to be unnecessary, and a verification model fit demonstrated that this assumption was valid, as $\hat{p}_{stack}(yes) + \hat{p}_{stack}(no) = 1$ for all observations.

Finally, the base classifier predictions and the linear model predictions were used to generate predictions on the DMC test set. Base SVM predictions were corrected using the method described in section 3.5.2.

3.5.4 Support vector regression

As an alternative level-1 regression model, support vector regression was performed using the caret and kernlab libraries [5, 38]. Just as in the previous section, the pooled order probability is the response variable and the base classifiers' predicted probabilities are the predictors.

A linear kernel was used. Tuning involved finding the optimal value of $0.25 \leq C \leq 1$ through 10-fold cross-validation. For a detailed description of the SVM tuning procedure, review section 3.4.2. The process is largely the same here; however, the dependent variable is continuous, and the measure to be optimized is the observed root-mean-square error (RMSE) between actual and predicted outcomes.

The tuning procedure demonstrated that there was no difference in RMSE due to changes in C; see Table A.5 for the table of results. Therefore, the default value of $C = 1$ was selected for the SVM regression model. As with the linear model, it was verified that $\hat{p}_{stack}(yes) + \hat{p}_{stack}(no) = 1$ for all observations, so no additional model was fitted for the 'no' class.

Finally, predictions were created on the DMC test set.

3.5.5 Evaluation issues

Ideally, we would now have performed an internal evaluation of the meta-classifiers using cross-validation. Unfortunately, due to time constraints this was not possible.

A time-consuming part of constructing the meta-classifier is the cross-validated train-and-predict loop which generates the level-1 data set (see section 2.5.5). In this process, all three base classifiers must be trained ten times so they can each generate predictions on their out-of-sample data.

To obtain 10-fold cross-validated estimates for the full *meta-classifier*, we would have to fit ten meta-classifiers, each on a subset of the training data. This means in total we would need to train $3 \times 10 \times 10 = 300$ base classifiers, which might require more than a week of computation time. It was decided to not pursue this procedure.

An alternative could be to train the meta-classifier on a subset of the training set only, and keep a held-out set for internal evaluation. However, this has the risks of penalizing the meta-classifier's performance through the smaller training set. It would remain the question if the test accuracy would really be comparable to the cross-validated accuracy estimates.

This leaves us with empirical evaluation of the meta-classifier on the test set. The actual classifications for the test set will be provided by DMC after the competition ends.

3.6 Transformation

In the DMC competition, participants can submit floating point predictions in the range $[0, 1]$ to signify the predicted probability of an order on a test session. We have therefore worked with class probabilities throughout our whole analysis pipeline.

The error function used by DMC to rank submissions is given in Equation 2.1. We have derived an accuracy measure from this function, the DMC accuracy, for use in our cross-validation procedures (see Equation 2.2). During these runs, it already became apparent that the number for cross-validated 'binary accuracy' (the default accuracy measure when predicting only binary classes) was always much higher than the average DMC accuracy.

Therefore, we simulated on a held-out test set how the DMC error would be influenced if we would submit not a classifier's order *probabilities* (predicted probabilities of 'yes' class membership), but just the values 1 or 0 based on the

most likely class.

The simulation showed that by just submitting 1 or 0 values, we decreased the DMC error on our submissions by a large margin. In fact, the DMC accuracy in this case was equivalent to the binary accuracy. It turns out, when using the DMC error function, it is much better to just submit binary predictions.

This can be demonstrated with a simple example. Consider a test set of 10 sessions on which we predict a 90% chance of ordering. Let's assume that in reality 9 of the 10 test cases led to an order. If we would submit probability predictions, the DMC error calculation would be as follows:

i	1	2	3	4	5	6	7	8	9	10
$order_i$	1	1	1	1	1	1	1	1	1	0
$prediction_i$	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
$\lvert order_i - prediction_i \rvert$	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.9

Table 3.8: DMC error calculation when submitting probabilities. The contributions to the DMC error $|order_i - prediction_i|$ are specified in Equation 2.1.

We can see that even the 'almost perfect' 90% order predictions contribute significantly to the DMC error, yielding a total $err_{DMC} = 1.8$. Now, if we would submit only binary predictions, the calculation looks as follows:

i	1	2	3	4	5	6	7	8	9	10
$order_i$	1	1	1	1	1	1	1	1	1	0
$prediction_i$	1	1	1	1	1	1	1	1	1	1
$\lvert order_i - prediction_i \rvert$	0	0	0	0	0	0	0	0	0	1

Table 3.9: DMC error calculation when submitting binary classes.

Now, most predictions do not contribute to the error, and the DMC error is only $err_{DMC} = 1.0$.

In short, *every* prediction except a 100% confident (and correct) prediction adds to the error function. There is no incentive to use binning translation functions, for instance to 'saturate' predictions towards their extremes while retaining the possibility of submitting floating point values for unsure cases. Submitting any non-integer values will only ensure that every observation will trigger an error penalty; it is better to just submit binary guesses while taking

the occasional penalty in case of a wrong prediction. This was further verified by simulating various bin sizes.

In conclusion, for the purpose of the DMC competition, it is optimal to submit binary class predictions only.

3.7 Final prediction

As a last step in the analysis, the various classifiers were executed on the DMC-provided test set [2]. As discussed, only the Random Forest classifications have been submitted to the DMC due to time constraints. Meanwhile, work continued on the meta-classifier.

The test set underwent the process sketched in Figure 2.2. First, features were extracted; then, missing values were imputed using unique-value imputation. Predictions were extracted from the tuned Random Forest, SVM and neural network base classifiers. These predictions were used as a test set for the stacked meta-classifier, which generated further predictions using linear model as well as linear support vector regression.

Classifier	RF	SVM	NN	stack LM	stack SVM
RF	1.000	0.924	0.784	0.996	0.990
SVM	0.924	1.000	0.789	0.928	0.935
NN	0.784	0.789	1.000	0.786	0.788
stack LM	0.996	0.928	0.786	1.000	0.994
stack SVM	0.990	0.935	0.788	0.994	1.000

Table 3.10: Prediction agreement between the various classifiers on the DMC test set. A value of 1 means the classifiers produced the exact same predictions on the test set.

To gather some ideas about the behavior of the stacking classifiers which we were not able to cross-validate, we created an agreement table in Table 3.10. The table shows that the stacked predictions do not diverge much from the predictions of Random Forest, which was selected as the best base classifier by cross-validation. The linear model stack differs on even less than 0.5% of the predictions, and the SVM stack changes just around 1% of the predictions. Possibly this represents a positive change, but this can only be evaluated using

actual order outcomes from the DMC test data.

3.8 DMC evaluation

On July 4th, the DMC organization released the actual outcomes of the test set on their website. [1] This allowed us to determine the real-world performance of all classifiers by comparing the predicted classes to the actual test outcomes. The results are presented in Table 3.11. Confusion matrices for all classifiers are given in Table 3.12.

Model	**CV accuracy**	**CV SD**	**Real accuracy**	err_{DMC}	**Rank**
Random Forest	0.898	0.005	0.903	496	1
Support vector machine	0.869	0.004	0.868	676	4
Neural network	0.778	0.027	0.755	1250	5
Stack LM	—	—	0.903	498	3
Stack SVM	—	—	0.903	496	2

Table 3.11: Cross-validated accuracy estimates of all classifiers on the training set, standard deviations of the cross-validated accuracy estimates, real accuracies on the DMC test set, and real DMC errors (see Equation 2.1) of the classifiers on the DMC test set. For classifiers submitting only binary predictions, the DMC error is the total number of errors on 5111 test observations.

It turns out that the stacking classifiers do not perform any better than the Random Forest. The SVM stack and the Random Forest have the same accuracy. If accuracy of classifiers is comparable, the most simple model must be preferred. [3: 4.4.4] Therefore, of our classifiers, Random Forest ranks first; the SVM stack follows, with the linear model stack following closely. As expected from cross-validation, the SVM classifier is somewhat less accurate, and finally the worst performing classifier is the neural network.

The cross-validated accuracy estimates are very similar to the real accuracies on the DMC test set, with the actual accuracies all falling within the cross-validated estimates plus/minus a single standard deviation. This shows that the cross-validated errors (and also Random Forest out-of-bag errors) are good estimates of the test error.

This concludes the presentation of our results. In the next chapter, we will interpret the results and review them in light of other work done on this prediction problem.

Random Forest	**actual no**	**actual yes**
no predicted	2502	212
yes predicted	284	2113

Support vector machine	**actual no**	**actual yes**
no predicted	2350	240
yes predicted	436	2085

Neural network	**actual no**	**actual yes**
no predicted	1835	299
yes predicted	951	2026

Stack LM	**actual no**	**actual yes**
no predicted	2496	208
yes predicted	290	2117

Stack SVM	**actual no**	**actual yes**
no predicted	2485	195
yes predicted	301	2130

Table 3.12: Confusion matrices of predicted versus actual order outcomes on the DMC test set.

Chapter 4

Discussion

4.1 Principal findings

The main problem of the DMC competition is to predict the probability that a web-shop visitor places an order, based on the transaction data given. As shown in section 3.8, the highest test accuracy of our classifiers was obtained by a Random Forest classifier having a test accuracy of 90.3%. This appears be a reasonable accuracy for practical purposes. To fully judge its merits however, it must be compared with the results of other teams; we will review our methods and results against those of the competition winner.

An important sub-question was which approach to use for dealing with the many missing values in the DMC data. As our results demonstrate (see Table 3.3), for this problem there is no significant difference between the methods. We find support for our intuition that in problems of pure prediction (not interpretation or calculation), extensive manipulation of the missing values does not give a more powerful result. Apparently imputation adds no information that a good classifier cannot itself derive from the remaining, present, variables. This implies that a simple imputation method, such as mean imputation or unique-value imputation, can be used safely in prediction problems. For a generic data mining problem, predictive imputation might be advantageous, but it requires computation time which in the case of pure prediction might be better spent elsewhere.

As a prediction model, Random Forest performed very well. Not only did it turn out to have the best cross-validated accuracy of our classifiers, it was also shown to have the best empirical test accuracy (90.3%, see Table 3.11). Besides ranking first of all our classifiers, it also has many other desirable

properties. First, the computational effort of training is relatively low when compared to SVM. Second, Random Forest only has one hyperparameter (m_{try}) to be defined externally, and its accuracy appears not even highly sensitive to the parameter, which confirms the author's assertions. [10] This means that if time is lacking, tuning might even be completely skipped and a Random Forest can just be trained with the default m_{try} setting used by the R randomForest package. Third, Random Forest's internal out-of-bag error estimate is validated as a quickly determined proxy for cross-validated error estimates. Thus, we can tune a Random Forest by simply training a small number of Random Forests with different values of m_{try} and comparing their out-of-bag errors. This fully relieves us from the computational expense of cross-validation, which (in the case of 10-fold cross-validation) would require us to train 10 classifiers for *each* hyperparameter setting. Finally, the ensemble architecture of a Random Forest allows for good class membership probability estimates, by averaging predictions over all trees. These properties make Random Forest very fast and useful for practical prediction, and we recommend Random Forest as a first try for a new problem.

The SVM classifier also performed with good accuracy (86.8% accuracy, see Table 3.11). However, it is associated with large practical drawbacks. First, its training times are much slower than Random Forest on large data sets. This problem is compounded by the requirement for cross-validation in order to estimate the test error, and even more by the larger number of hyperparameters to be selected and tuned (the kernel function, the kernel's parameters, and the cost parameter C). Tuning therefore requires us to cross-validate an already large number of SVMs. Taking into account all the added computation effort, Random Forest can produce comparable (or better) results in a fraction of the time. Of course, it would be advisable to always create an additional SVM classifier, in order to validate that Random Forest's accuracy is competitive on a particular data set, but this might be done in parallel with other analysis steps.

The neural network has an unacceptably low accuracy for this problem. This might be expected due to the simple network topology with a single hidden layer. Perhaps, more complex neural networks would have been able to model the patterns in the training data more adequately. In an attempt to explain the poor

performance of the model in relation to network topology, we have attempted to train more elaborate models, such as a multi-layer perceptron using the RSNNS library [44], but these training attempts did not converge and resulted in networks with accuracies comparable to random guessing. This is likely due to the large number of hyperparameters that must be selected correctly. We conclude that neural networks have theoretical significance, but their behavior is fragile and their utility in practical prediction seems limited when compared to more modern alternatives.

Unfortunately, the stacking meta-classifier does not yield any improvements in test accuracy for our problem. This is true for the standard linear model stack as described by its author, [24] but also for the SVM stack as used for instance in the MetaFraud framework [27]. It is likely that stacked generalization would have worked better on other data sets. We used the same architecture as Ting and Witten [25] with three base classifiers, a probability-based level-1 data set, and multi-response linear regression (MLR). In their empirical evaluation of stacking, they obtained higher test accuracies relative to selecting the best base classifier by cross-validation on many data sets. Even recently, a stacking approach was very successful as it reached the second place of the Netflix competition. [45] For our problem, an advantage of stacking was not realized.

The disappointing performance of the stack might have been due to the fact that stacking produces a weighted combination of base classifiers for every class (review Equation 3.3), for instance, giving more weight to base classifier 1 for class c_i, but more weight to base classifier 2 for class c_j. But, in a two-class problem, after the weightings for a single class are determined, there are no degrees of freedom left. In this case, it might not be surprising that the stack just largely followed the predictions made by the best base classifier with marginal changes. There are extensions to stacking, such as the use of entropy-based level-1 attributes and multi-response model trees, which have shown to give better results than normal stacking in many cases. [23] However, it is hard to say if advantages will be realized in this particular type of problem with few classes, few base classifiers, and a clear 'best' base classifier. We suspect that stacking may produce better results in multi-class problems or when using more base classifiers.

Due to the lack of an available R library, implementing the stacking algorithm took a comparatively long time. The level-1 training set generation is also computationally very expensive, which makes cross-validation of the stack intractable (see section 3.5.5). This makes stacking unattractive for this type of problem.

We chose to use only open-source software for the analysis. This turned out to be viable. R was demonstrated to be a powerful environment for data analysis. Its code-oriented nature enables external validation and reproduction of methods and results. Important drawbacks of R are the lack of parallelism and the lack of standardization in third-party libraries, as explained in section 3.1. Both drawbacks can be mitigated to a large degree by using the right combination of packages. It turns out that the caret package [5], when used with the foreach [31] and doMC [32] packages, is a highly valuable tool to cross-validate and tune prediction models in a parallel fashion. It also provides a standardized interface for many modern prediction algorithms. There are various other packages implementing models, some of which are redundant. When searching for a prediction algorithm in R, it is recommended to first check whether it is offered by the caret package.

For general non-statistical programming tasks, such as text file processing, data acquisition and data conversion, the R language with its long pedigree and sometimes arcane syntax is less appropriate. Languages such as Perl, Python and PHP are more current in regards to object oriented programming, text processing, input/output facilities, error handling, documentation and examples, et cetera.

4.2 Context

In the competition's results, our Random Forest classifier (with an err_{DMC} of 496) ranked 27th out of 63 competitors (see Table A.6 for the full ranking). Our result is modestly better than the median err_{DMC} of 607. The best submission had an err_{DMC} of 144 (97.2% accuracy) and the lowest ranking submission which is not an outlier had an err_{DMC} of 2772 (45.8% accuracy). The absolute worst valid submission had an err_{DMC} of 4970.22 out of 5111 observations (2.75% accuracy), which looks rather tragic, as it seems to be a case of switched

yes/no labels — if the team would have submitted the complements of their predictions, they would have won the competition with an err_{DMC} of 140.78 (97.25% accuracy).

After the competition, we have discussed approaches with the winning team of the Technische Universität Dortmund. Interestingly, there are many similarities in approach. We summarize the techniques and their differences in Table 4.1.

Aspect	**Our method**	**Winning method**
Feature extraction	62 features, aggregated using count, sum, average, max, min, range, standard deviation, gradient; exclude customerId	160 features, aggregated using min, max, mean, standard deviation, last non-missing value, difference between two last non-missing values
Imputation	unique-value imputation with –1 as imputed value	unique-value imputation with $2 \times max(x_i)$ as imputed value
Prediction	stack of Random Forest, SVM, and neural network	bagging of 600 C4.5 decision trees
Transformation	predict binary class only	predict binary class only

Table 4.1: Comparison of data mining approaches.

The winning team used bagging to create an ensemble of 600 C4.5 decision trees. This can be seen as manually creating a Random Forest-like classifier without de-correlation of trees (see section 2.5.2).

Our best performing prediction models (Random Forest, SVM, and the stack) produced test accuracies falling roughly in the same range (around 90%, see Table 3.11). This leads us to the conjecture that implementing another prediction model, such as bagging C4.5 decision trees, might not bring huge accuracy improvements to our data set, especially since the 600 C4.5 decision tree ensemble and Random Forest are already very similar.

Instead, we speculate that the difference in accuracies is largely due to a difference in feature extraction. For instance, we did not use a 'most recent value' aggregation, which could have been very informative: for instance the last product prices viewed, or the most recent step in the order process. We also ignored the identificator variable *customerId* as we thought it might produce overly optimistic error estimates as its information would not generalize to new customers, but we did not explore if extracting a customerId feature might have been beneficial in reducing DMC error (for instance, if customers overlap

between training and test set).

Feature extraction seems to be at least partly a creative process. At various moments in time, we have added new features or data aggregation operations after exploring some part of the training data. Therefore, as the experience with a data set grows, the full data mining pipeline should be amenable to adding new features and re-doing the steps after feature extraction. R's procedural programming nature makes it relatively easy to re-trace steps on new data. Unfortunately, some of our analysis components, such as the SVM and stacking classifiers, required a training time measured in days, which made it impractical to experiment with new features and iterate.

In many situations, the analyst could 'prototype' a model on a subset of the training data to speed up the process, but in a competition setting where every bit of accuracy matters, it is preferable to work with the full data set as much as possible, in order to prevent making decisions on biased subset estimates.

This underscores the importance of a classifier such as Random Forest that can be tuned quickly, and that provides internal test error estimates to prevent the need for expensive cross-validation.

While for practical use of the order prediction models, predicting the order *probability* might be useful, the DMC error function was minimized by only predicting binary classes. This shows that if a prediction problem is given, it pays off to check the behavior of the error function carefully and possibly adjust predictions.

4.3 Interpretation

We have created multiple prediction models that successfully predict the probability of an order. It can be informative to explore which predictors play a large role in prediction.

A drawback of Random Forest and SVM classifiers is the lack of interpretability of their resulting models, which consist of large amounts of structured data (for Random Forest, a large number of decision trees; for SVM, a set of support vectors in 62-dimensional feature space). To explore the structure of the decision problem, it is easier to use a singular decision tree. We refer to the decision

trees we have trained in Figure 3.1, in particular, the decision tree trained on unique-value-imputed data denoted with (c). A description of DMC transaction variables is given in Table 2.1, from which we have extended session features as described in section 3.2.

All decision trees agree that *maxBStep* is the most informative feature. This feature represents the maximum step reached in the order process by the visitor. The ordered levels are not described in the DMC set, but we can assume that they consist of steps such as: product selection, address entry, confirmation, et cetera. Visitors not reaching a high order step in the session are not likely to place an order. The feature *avgBStep* (average step in order process over all transactions) at the bottom of tree (c) plays a similar role.

Features such as *avgBCount* (average number of products in the shopping basket), *sumDuration* (the total time that the visitor spent on the web-shop), and *countTransact* (number of transactions by the visitor) work in a similar fashion. If a visitor adds many products to their shopping basket, or if they use the web-shop for a long time, they are more likely to place an order.

Some customer-related attributes also appear to have predictive value. For instance, from tree (c) we find that customer *age* matters, with customers of 19 years and younger not likely to place an order. Note that in the unique-value imputation tree, customers with no known age have been imputed with the value –1 and thus also are viewed as less likely to order. In the two other trees, these customers are lumped together with customers of average age, and this may explain why the feature *age* does not appear in those trees.

Another customer-related attribute is the feature *payments*, which is used in trees (a) and (b) in place of age. This feature denotes the number of payments made earlier by the customer. Interestingly, a lower number of payments gives rise to a higher order probability.

These features were apparently rated as important by the decision tree algorithms. We can relate these to the variable importance measures generated by Random Forest in Figure 3.3. Indeed, we find that many features used in the decision trees, such as *maxBStep*, *avgBStep*, and *countTransact*, are rated as high-importance in the Random Forest. Other features are less prominent.

Some features ranked as important by Random Forest, but not by the decision

tree algorithm, are *maxAvailability* (or the maximum level of availability for the products viewed, based on the ordered categorical 'availability' variable which ranges from 'no products in stock' to 'all products in stock') and *address* (which distinguishes between companies, male and female customers).

While decision trees are less stable and accurate than Random Forests, they are of great importance in exploring the structure of a decision problem. When feature space is of low dimension, prediction contour plots such as Figure 2.7 are highly useful, but in our case with 62 features, interpretability by plotting is low. Therefore we recommend to train and plot a decision tree at the start of a prediction problem, for the purpose of exploring relationships of features with the outcome.

4.4 Limitations

In the former section, we have derived from decision trees and Random Forest some important features and structure of the prediction problem. A valid question may be if these predictors really give us any new insight on the data.

Among the most important features are *maxBStep* (maximum step in the order process reached), *avgBCount* (average number of products held in basket), *countTransact* (number of transactions), *sumDuration* (total on-line time), and *maxAvailability* (maximum availability of the products viewed). But the significance of these features could be seen as somewhat trivial — It is only expected that a visitor who spends a long time on a web-shop, views many pages, or selects to put many products in their shopping basket, is more likely to order in that session. However, this information is only available when the visitor has spent a lot of time in the web-shop, or when the session has concluded.

So, most of the important predictors that help a classifier perform well in this DMC task, actually do not give us much information in assessing the order probability of a new visitor and customizing the web-shop experience for them in time. In a way, the problem suffers from its definition: competitors are asked to predict a property of a session, which is largely influenced by predictors which are only fully available *after* that session has concluded. The practical relevance of the high prediction accuracies seen in the competition is therefore

questionable.

DMC task 2, the on-line prediction task (review section 1.2), might produce classifiers that bring wider insights. In this task, agents must make stream-based predictions of transactions that arrive in real-time. Thus, an agent must generate a prediction after every transaction, based on the growing body of information about the current session that is constantly amended by new transactions. In this task, a classifier that only makes accurate predictions using late-generated information (such as the maximum order step) would have a higher error than a classifier that also predicts well using only a priori information.

That is not to say that the classifiers created in DMC task 1 are worthless in a real-life situation. If we were to use a classifier to predict order outcomes at the start of a session *before* transactions take place, we might simply consider all dynamic session-based features as missing values, or create a separate model trained only on a feature subspace containing only customer-based features such as age, plus globally available information such as weekday, hour, et cetera. For instance, if we train a Random Forest using only customer information, weekday and hour, we get an estimated accuracy of 69.2%. This means the classifier still has some predictive ability, even if the major 'trivial' features such as a visitor's ordering step are not available.

4.5 Future work

As the competition demonstrates, there is definitely room for improvement to our best classifier, for instance through improved feature extraction. As discussed, feature extraction is to some extent a creative process, but there is also considerable theory and a large body of continuously evolving practices attached to it. The effort in our work was mostly focused at creating better imputation and prediction models; it is therefore an important realization that ultimately a highly simple imputation method and a popular prediction algorithm, namely unique-value imputation and a Random Forest with standard parameters, performed very well at their respective tasks – while more improvements would likely have been realized by focusing on improving feature extraction. For future studies, it would therefore be beneficial to perform a systematic review of

applicable feature extraction techniques early in the process. Review works such as Guyon and Elisseeff [46] may provide necessary guidance as to which types of techniques would work well in a certain situation.

While we conclude that the choice of various imputation techniques did not affect the accuracy of our classifiers, it might be possible to improve accuracy by a 'reduced model' technique suggested by Saar-Tsechansky and Provost [4]. The technique, which is not widely used today, depends on training multiple 'reduced models' which include only a subset of the features. When encountering a test observation having missing values, it is predicted using the model that fits the known variables for that observation. The general idea is that a model's internal structure and optimal hyperparameters associated with the full feature space, might not be optimal for prediction on a subspace of the feature space. The authors demonstrate empirically that the reduced model technique produces a lower bias than either simple or advanced imputation techniques on many data sets. Naturally, fitting many models increases training time, although this can be mitigated by using a fast classifier such as Random Forest. There are also compromises possible in order to decrease computation time, for example, create only reduced models having missing value sets occurring heavily in the data set, and use simple (mean/unique-value) imputation techniques for other observations.

It would also be interesting to make a detailed comparison between Random Forest and the bagged decision tree classifier as used by the winning team in the DMC competition, perhaps on other data sets. Due to the similarity of classifier structure, we might expect both classifiers to have similar accuracies, but it remains an open question if the winner's bagging classifier without de-correlation might bring accuracy improvements over Random Forest in certain cases.

In conclusion, the obtained accuracy by our methods seems reasonable and lies above the median result of the competition. However, by restricting the prediction model to the Random Forest classifier or a similar model, expanding feature extraction, and implementing a reduced model technique, it is possible that significant accuracy improvements could be obtained.

Chapter 5

Conclusions

After tuning three base classifiers on the DMC training set, we find that the Random Forest classifier has the lowest test error, with a 90.3% accuracy on the DMC test set. The support vector machine classifier reached 86.8% accuracy. The neural network performed worst with 75.5% accuracy. Random Forest and SVM can both be recommended as general purpose classifiers. However, the Random Forest method was shown to have many desirable properties, such as relatively fast training and a low number of external model parameters to be defined. Its internal out-of-bag error estimates are highly useful for time-limited prediction problems such as competitions, since they allow for test error estimation without long cross-validation computations. Therefore we highly recommend Random Forest as a 'gold standard' to use as a first step on a new prediction problem. We recommend to also train and plot decision trees for exploration and interpretation of a prediction problem.

In the DMC competition, our best classifier ranked 27th out of 63 competitors. The test accuracy obtained by the competition winner was 97.2%. This demonstrates that order prediction from web-shop transaction logs is possible with high reliability. However, the high accuracies in the competition were only achieved by using transaction features available after the user session has concluded. This limits the practical relevance of the high accuracies for predicting the behavior of a new visitor. Still, we show that there is predictive power in that situation, for instance by using features derived from a customer database, such as customer age, number of earlier payments by the customer, et cetera.

In case the analysis depends on a choice in preprocessing (such as the choice of imputation approach), error estimates and decision tree plots should be used

to verify if prediction model accuracy and structure is sensitive to this choice. When handling missing data, we found no empirical differences in classifier accuracy for various imputation methods such as mean imputation, predictive imputation using boosted regression trees, or an uncommon technique called unique-value imputation. This demonstrates that a simple and fast imputation technique, such as unique-value imputation, is acceptable for use in prediction problems. In a prediction setting, there appears to be no value in predictive imputation.

We have implemented a meta-classifier using stacked generalization that combines the results of Random Forest, SVM and neural network classifiers. We found empirically that, on the DMC problem, the stacking classifier did not perform better than Random Forest. We trained stacking classifiers using ordinary least squares linear regression and support vector regression with similar results. The changes in predictions relative to Random Forest were minimal and did not increase accuracy. A practical disadvantage of stacking is the very long computation time associated with generating the stack's training set, as this generation is based on cross-validation. This makes the data analysis process less agile and complicates creative iteration on earlier process steps. It also makes cross-validation of the full stacking classifier impractical, as it would take computation time equivalent to 300 base model training times. The lack of a practical test error estimate brings considerable uncertainty. There are extensions to stacking, but it is unsure if the possible benefits would be worth the cost. Therefore, we do not recommend the use of stacking in future problems similar to this study, although in other situations it can be a useful technique.

We successfully built all classifiers using the open source package R as a statistical environment. The growing popularity of R in data mining circles is definitely justified. Most modern prediction algorithms are available as R packages. R allows for easy code sharing and reuse, which improves reproducibility of research. The caret package is one of the most important packages for predictive modeling, as it provides a standardized interface for many prediction models, and allows for fast model tuning on multi-processor systems. For non-statistical programming tasks, such as text processing and data preparation, we recommend using a more modern general-purpose scripting language, such as Perl,

Python or PHP. We conclude that the R environment, in conjunction with the caret package and a scripting language for input processing, constitutes a very useful platform for data mining and predictive modeling in particular.

Bibliography

[1] prudsys Data Mining Cup Competition 2013. http://www.data-mining-cup.de/en/dmc-competition/, 2013.

[2] Competition Task, prudsys Data Mining Cup Competition 2013. http://www.data-mining-cup.de/en/dmc-competition/task/, 2013.

[3] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley Longman, Boston, USA, 1st edition, 2005.

[4] Maytal Saar-Tsechansky and Foster J. Provost. Handling missing values when applying classification models. *Journal of Machine Learning Research*, 8:1623–1657, 2007.

[5] Max Kuhn. Building predictive models in R using the caret package. *Journal of Statistical Software*, 28(5):1–26, 11 2008.

[6] Bradley Efron and Robert Tibshirani. Improvements on cross-validation: the 632+ bootstrap method. *Journal of the American Statistical Association*, 92(438):548–560, 1997.

[7] Trevor Hastie, Robert Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. New York: Springer-Verlag, 2nd edition, 2009.

[8] Brian Ripley. *tree: Classification and regression trees*, 2012. R package version 1.0-33.

[9] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[10] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[11] Andy Liaw and Matthew Wiener. Classification and regression by randomForest. *R News*, 2(3):18–22, 2002.

[12] Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis, and Torsten Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinformatics*, 8(1):25, 2007.

[13] Mark R. Segal. Machine learning benchmarks and random forest regression. http://www.epibiostat.ucsf.edu/biostat/cbmb/publications/bench.rf.regn.pdf, 2003.

[14] Alexandros Karatzoglou, David Meyer, and Kurt Hornik. Support vector machines in R. *Journal of Statistical Software*, 15(9):1–28, 4 2006.

[15] David Meyer. Support vector machines; the interface to libsvm in package e1071. http://stuff.mit.edu/afs/athena.mit.edu/software/r/current/arch/i386_ubuntu1104/lib/R/library/e1071/doc/svmdoc.pdf, 2012.

[16] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[17] Asa Ben-Hur and Jason Weston. A users guide to support vector machines. In *Data mining techniques for the life sciences*, pages 223–239. Springer, 2010.

[18] Qun Chang, Qingcai Chen, and Xiaolong Wang. Scaling Gaussian RBF kernel width to improve SVM classification. In *Neural Networks and Brain, 2005. ICNN&B'05. International Conference on*, volume 1, pages 19–22. IEEE, 2005.

[19] Chrislb, Wikipedia. Diagram of an artificial neuron. https://upload.wikimedia.org/wikipedia/commons/6/60/ArtificialNeuronModel_english.png, 2005. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license.

[20] Marcus W. Beck. Visualizing neural networks from the nnet package. R-bloggers, http://www.r-bloggers.com/visualizing-neural-networks-from-the-nnet-package/, 2013.

[21] R. Callan. *The Essence of Neural Networks*. The Essence of Computing Series. Prentice Hall Europe, 1999.

[22] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992.

[23] Saso Džeroski and Bernard Ženko. Is combining classifiers with stacking better than selecting the best one? *Machine learning*, 54(3):255–273, 2004.

[24] David H. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[25] Kai Ming Ting and Ian H Witten. Issues in stacked generalization. *Journal of Artificial Intelligence Research*, 10:271–289, 1999.

[26] Sam Reid and Greg Grudic. Regularized linear models in stacked generalization. In *Multiple Classifier Systems*, pages 112–121. Springer, 2009.

[27] Ahmed Abbasi, Conan Albrecht, Anthony Vance, and James Hansen. MetaFraud: a meta-learning framework for detecting financial fraud. *MIS Quarterly*, 36(4):1293–1327, 2012.

[28] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012.

[29] Darrel C. Ince, Leslie Hatton, and John Graham-Cumming. The case for open computer programs. *Nature*, 482(7386):485–488, February 2012.

[30] Jon Hill, Matthew Hambley, Thorsten Forster, Muriel Mewissen, Terence Sloan, Florian Scharinger, Arthur Trew, and Peter Ghazal. SPRINT: a new parallel framework for R. *BMC Bioinformatics*, 9(1):558, 2008.

[31] Revolution Analytics. *foreach: Foreach looping construct for R*, 2012. R package version 1.4.0.

[32] Revolution Analytics. *doMC: Foreach parallel adaptor for the multicore package*, 2013. R package version 1.3.0.

[33] PHP: Hypertext Preprocessor. http://www.php.net/, 2013.

[34] Jeffrey Wong. *imputation*, 2013. R package version 2.0.1.

[35] Murat Sariyar, Andreas Borg, and Klaus Pommerening. Missing values in deduplication of electronic patient data. *Journal of the American Medical Informatics Association*, 19(e1), 2012.

[36] Max Kuhn. Contributions from Jed Wing, Steve Weston, Andre Williams, Chris Keefer, Allan Engelhardt, and Tony Cooper. *caret: Classification and Regression Training*, 2013. R package version 5.15-61.

[37] Andreas Alfons. *cvTools: Cross-validation tools for regression models*, 2012. R package version 0.3.2.

[38] Alexandros Karatzoglou, Alex Smola, Kurt Hornik, and Achim Zeileis. kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9):1–20, 2004.

[39] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, USA, 4th edition, 2002.

[40] Zach Mayer. caretEnsemble: Framework for combining caret models into ensembles. https://github.com/zachmayer/caretEnsemble, 2013.

[41] John Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

[42] Joseph Drish. Obtaining calibrated probability estimates from support vector machines. 1998.

[43] Brian S. Everitt and Graham Dunn. *Applied multivariate data analysis*. Arnold, London, UK, 2nd edition, 2001.

[44] Christoph Bergmeir and José M. Benítez. Neural networks in R using the Stuttgart neural network simulator: RSNNS. *Journal of Statistical Software*, 46(7):1–26, 2012.

[45] Joseph Sill, Gábor Takács, Lester Mackey, and David Lin. Feature-weighted linear stacking. *arXiv preprint arXiv:0911.0460*, 2009.

[46] Isabelle Guyon and André Elisseeff. *Feature extraction: foundations and applications*, volume 207. Springer, 2006.

Appendix

A.1 Random Forest tuning table

m_{try}	**Binary accuracy**	**DMC accuracy**	**SD Binary accuracy**	**SD DMC accuracy**
2	0.883	0.766	0.005	0.010
10	0.896	0.792	0.005	0.010
19	0.898	0.795	0.005	0.010
28	**0.898**	**0.796**	**0.005**	**0.011**
36	0.898	0.795	0.005	0.011
45	0.897	0.794	0.006	0.011
54	0.897	0.794	0.005	0.011
63	0.897	0.793	0.005	0.010

Table A.1: Detailed results of tuning Random Forest to optimize accuracy using cross-validation. The best estimated binary and DMC accuracies are both provided by $m_{try} = 28$.

A.2 Bootstrapped SVM tuning table

C	σ	**Binary accuracy**	**DMC accuracy**	**SD Binary accuracy**	**SD DMC accuracy**
0.25	0.01	0.853	0.707	0.002	0.004
0.25	0.1	0.856	0.711	0.002	0.004
0.25	1	0.734	0.484	0.003	0.006
0.25	10	0.613	0.265	0.006	0.009
0.5	0.01	0.855	0.711	0.002	0.005
0.5	0.1	0.861	0.721	0.002	0.004
0.5	1	0.762	0.536	0.003	0.005
0.5	10	0.633	0.300	0.004	0.006
1	0.01	0.858	0.716	0.002	0.004
1	**0.1**	**0.863**	**0.726**	**0.002**	**0.004**
1	1	0.776	0.561	0.003	0.006
1	10	0.641	0.315	0.004	0.005
2	0.01	0.861	0.723	0.002	0.005
2	0.1	0.863	0.725	0.002	0.004
2	1	0.778	0.564	0.003	0.006
2	10	0.645	0.320	0.004	0.005

Table A.2: Detailed results of first SVM tuning attempt using 25-repeated bootstrap. Hyperparameters selected were $C = 1$ and $\sigma = 0.1$.

A.3 Sigma-optimized SVM tuning table

C	**Binary accuracy**	**DMC accuracy**	**SD Binary accuracy**	**SD DMC accuracy**
0.25	0.855	0.710	0.004	0.008
0.5	0.863	0.726	0.004	0.007
1	**0.869**	**0.738**	**0.004**	**0.008**

Table A.3: Detailed results of second SVM tuning attempt using 10-fold cross-validation and σ held constant at $\sigma = 0.0371$. Hyperparameter selected was $C = 1$.

A.4 Neural network tuning table

size	decay	Binary accuracy	DMC accuracy	SD Binary accuracy	SD DMC accuracy
1	0	0.769	0.546	0.023	0.044
1	0.01	0.741	0.493	0.035	0.066
1	0.1	0.757	0.522	0.030	0.056
2	**0**	**0.778**	**0.561**	**0.027**	**0.052**
2	0.01	0.765	0.535	0.025	0.050
2	0.1	0.761	0.525	0.023	0.049
3	0	0.763	0.527	0.017	0.035
3	0.01	0.762	0.525	0.014	0.029
3	0.1	0.772	0.546	0.012	0.027
4	0	0.768	0.537	0.016	0.032
4	0.01	0.767	0.532	0.017	0.036
4	0.1	0.765	0.532	0.013	0.029
5	0	0.766	0.536	0.016	0.032
5	0.01	0.764	0.530	0.021	0.040
5	0.1	0.758	0.517	0.022	0.046
6	0	0.773	0.547	0.014	0.031
6	0.01	0.760	0.524	0.016	0.029
6	0.1	0.770	0.543	0.016	0.034
7	0	0.769	0.540	0.023	0.047
7	0.01	0.761	0.522	0.017	0.034
7	0.1	0.760	0.519	0.018	0.041
8	0	0.765	0.533	0.021	0.046
8	0.01	0.766	0.534	0.018	0.037
8	0.1	0.764	0.531	0.015	0.030
10	0	0.760	0.522	0.013	0.026
10	0.01	0.761	0.524	0.021	0.043
10	0.1	0.763	0.527	0.018	0.038
12	0	0.770	0.542	0.021	0.042
12	0.01	0.770	0.544	0.016	0.030
12	0.1	0.772	0.546	0.014	0.028

Table A.4: Detailed results of neural network tuning for size (number of hidden nodes) and weight decay. Standard deviations are relatively large. Hyperparameters selected are $size = 2$ and $decay = 0$.

A.5 Linear SVM meta-classifier tuning table

C	**RMSE**	R^2	**SD RMSE**	**SD** R^2
0.25	0.282	0.688	0.005	0.010
0.5	0.282	0.688	0.005	0.010
1	0.282	0.688	0.005	0.010

Table A.5: Detailed results of linear SVM meta-classifier tuning using 10-fold cross-validation. As the table shows, the accuracy does not depend on the choice of C.

A.6 Team rankings of DMC competition

Rank	Team	err_{DMC}	Rank	Team	err_{DMC}
1	TU_Dortmund_2	144.00	33	FH_Frankfurt_1	667.00
2	TU_Dortmund_1	145.00	34	THS_Mittelhessen_2	703.00
3	Inst_Karlsruhe_1	146.00	35	THS_Mittelhessen_1	752.00
4	Inst_Karlsruhe_2	150.05	36	HS_Hannov er_1	792.50
5	Uni_Iowa_State_1	154.00	37	HS_Mittweida_1	814.00
6	Uni_Athen_EB_1	158.00	38	Uni_Hamburg_1	829.26
7	Uni_Marburg_1	162.00	39	Uni_Cyril_Methodius_Skopje_1	880.00
8	FZI_Karlsruhe_2	165.00	40	Uni_Koeln_1	892.00
9	Inst_Bandung_Technology_2	166.55	41	Uni_Paderborn_1	935.00
10	Inst_Bandung_Technology_1	168.00	42	Uni_Jagiellonian_1	1006.00
11	Uni_Siberian_State_Aerospace_1	168.00	43	FH_Frankfurt_2	1023.00
12	FZ_Desy_1	173.00	44	TU_Berlin_1	1069.00
13	TU_Amirkabir_1	185.76	45	Uni_Southern_California_1	1091.62
14	Uni_AGH_ST_1	192.00	46	Uni_Indonesia_1	1275.00
15	Uni_Pendidikan_1	200.00	47	Uni_Tehran_1	1455.30
16	Uni_Budapest_TE_1	211.00	48	Uni_Tehran_2	1556.60
17	Uni_Pernambuco_1	224.12	49	Uni_Iran_ST_1	1728.00
18	HWR_Berlin_1	257.00	50	Uni_Shahid_Beheshti_1	1797.00
19	Uni_Perm_Polytechnic_1	285.00	51	Uni_Warwick_1	1898.99
20	HS_Anhalt_2	289.00	52	Uni_Budapest_TE_2	1929.00
21	Uni_Buenos_Aires_1	293.00	53	Uni_Cyril_Methodius_Skopje_2	1980.46
22	FH_Brandenburg_1	314.00	54	Uni_Sakarya_1	2329.00
23	HS_Hannov er_2	359.48	55	Uni_Indonesia_Education_1	2446.00
24	FZI_Karlsruhe_1	425.00	56	Uni_California_1	2590.92
25	TU_Darmstadt_2	451.00	57	School_Walker_Governor_1	2615.53
26	TU_Darmstadt_1	493.00	58	Uni_Indonesia_2	2744.00
27	Uni_Ersamus_Rotterdam_1	496.00	59	Uni_Queens_1	2772.00
28	School_Economics_Warsaw_1	514.00	60	Uni_Aristotle_Thessaloniki_1	4970.22
29	School_Economics_Warsaw_2	522.00	61	Uni_Northwestern_1	NA
30	Uni_Cairo_1	530.00	62	Uni_Perm_Polytechnic_2	NA
31	FH_Muenster_1	607.00	63	School_Walker_Governor_2	NA
32	HS_Anhalt_1	646.00			

Table A.6: Rankings and DMC error values (see Equation 2.1) for all teams in the competition.