–

Development of a predictive model for the classification of products

Case Study

Frankfurt School of Finance & Management

Simon Prey, Franziska Pilz, David Bedoian

Table of Contents

1 Introduction

The E-commerce sector has seen tremendous growth in recent years through the increasing use of mobile devices and widespread access to broadband internet. Alongside these developments E-commerce stores has been able to establish themselves as credible competitors to traditional forms of retail, leading to an ever-steady growth in the online consumer base. Recent developments have only intensified the shift towards online purchasing. Within the sector of E-commerce, online stores strive to provide customers with the best shopping experience possible to differentiate themselves from their competitors. This includes steady improvements to the online storefront and its usability.
Studies show that 76%[1] of consumers perceive "ease of use" as the most important characteristic a website can possess, meaning that the website has to make it easy for the customer to find the products they are looking for. Therefore, it is crucial to make searching for a desired product quick and simple. A major step to improving this search functionality is taken by classifying products listed for sale into individual categories. This allows for more straightforward browsing of products via categories and allows algorithms to recommend products to shoppers based on similar categories.

The Otto Group is one of the largest online retailers in the world which offers around three million different products on its website and smartphone application. The online segment of the Otto Group currently creates an annual revenue of 8,1 billion euros[2]. With a constantly increasing product portfolio, classifying the products individually into their respective categories requires a lot of manual labor. A alternative way may be found by creating a classifier trained on historical product data that can accurately classify new products added to the online store.

1.1 Problem Outline

In this case study, the challenge was to classify more than 200,000 products from Otto Group into nine main product categories. The objective is to explore classification models in order to

[1] Gary, H. (2019, 14. Juli). 15 Must-Have Features for Ecommerce Sites. Retrieved from Search Engine Journal: https://www.searchenginejournal.com/ecommerce-guide/must-have-website-features/#close

[2] Otto Group. (2020). 2019 annual report. https://www.ottogroup.com/en/about-us/daten-fakten/Annual_Reports.php

build a predictive model to best classify products into their respective categories for further business analysis and decision-making. This is crucial for a big online retailer as OTTO since they have constantly changing product offerings that require constant automatic classification. In this paper, different classification approaches are tested in terms of their accuracy and hence their applicability to the challenge.

2 Exploratory Analysis and Data Preparation

After getting familiar with the company and identifying its corresponding business problem, its crucial to get a comprehensive overview of the dataset provided. The main objective of this section is to set a representative foundation on which the classification model can be built. For this purpose, it is important to prepare the dataset and to make sure that it does not have any missing values, potential outliers and is properly balanced to avoid weighting errors and other misclassifications.

2.1 Exploratory Data Analysis

After setting the working directory, the dataset is read into R through the **read.csv()** function.

```
# Setup
setwd("C:/Users/simon/Desktop/Group Project")
data = read.csv("ClassifyProducts.csv")
```

As a next step, the **str()** function is used to get an impression of the structure of the dataset and the type of the variables that are available. As seen below, the dataset consists of around 62.000 observations divided in 95 variables with one portraying the product ID, another one as the target variable and 93 product features.

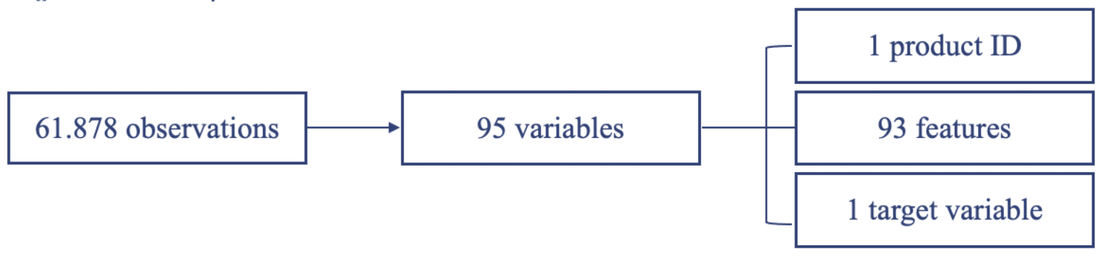


Figure 1: Dataset - Structure

The target variable defines the nine product classes in which new products should be classified based on the 93 features. The product features and their characteristics are not further defined.

```
str(data)

## 'data.frame':    61878 obs. of  95 variables:
##  $ id     : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ feat_1 : int  1 0 0 1 0 2 2 0 0 0 ...
##  $ feat_2 : int  0 0 0 0 0 1 0 0 0 0 ...
##  $ feat_3 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ feat_4 : int  0 0 0 1 0 0 0 0 0 0 ...
##  $ feat_5 : int  0 0 0 6 0 7 0 0 0 0 ...
```

To determine the exact number of observations within each category, the **table()** function is applied. This step makes it possible to see the distribution of the data within each class which is relevant for data preparation at a later stage. The *ggplot* library is also applied to better visualize the distribution of the variables.

```
table(data$target)

##
## Class_1 Class_2 Class_3 Class_4 Class_5 Class_6 Class_7 Class_8 Class_9
##    1929   16122    8004    2691    2739   14135    2839    8464    4955

ggplot(data = data, aes(x = target)) +
  geom_bar(fill="steelblue")+
  ggtitle("Distribution of Target Variable")+
  theme_minimal()
```

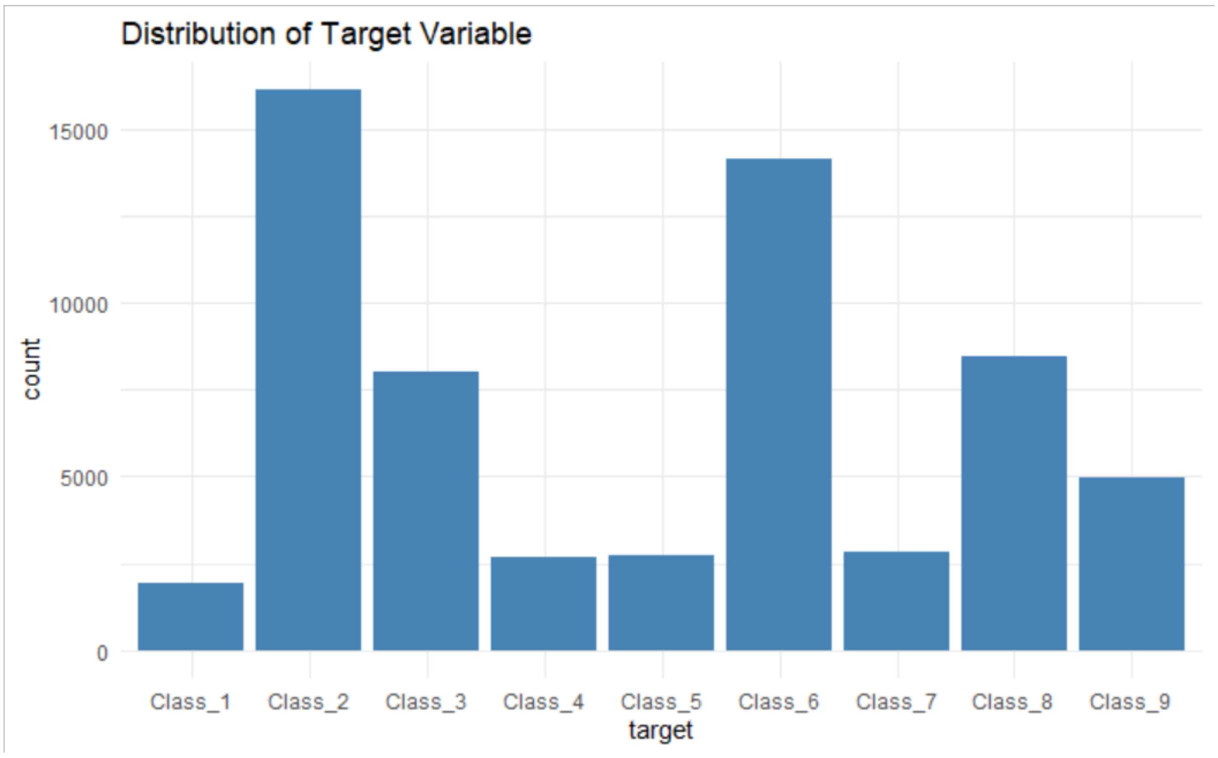


Figure 2: Observation Distribution of Target Variable

The **summary()** function is then used to get a more detailed view of each feature, including the minimum, maximum and mean value. The output of the function shows the range of the features and helps to evaluate the overall scope of the data.

```
summary(data)

##        id            feat_1            feat_2            feat_3
##  Min.   :    1   Min.   : 0.0000   Min.   : 0.0000   Min.   : 0.0000
##  1st Qu.:15470   1st Qu.: 0.0000   1st Qu.: 0.0000   1st Qu.: 0.0000
##  Median :30940   Median : 0.0000   Median : 0.0000   Median : 0.0000
##  Mean   :30940   Mean   : 0.3867   Mean   : 0.2631   Mean   : 0.9015
##  3rd Qu.:46409   3rd Qu.: 0.0000   3rd Qu.: 0.0000   3rd Qu.: 0.0000
##  Max.   :61878   Max.   :61.0000   Max.   :51.0000   Max.   :64.0000
```

2.2 Data Preparation

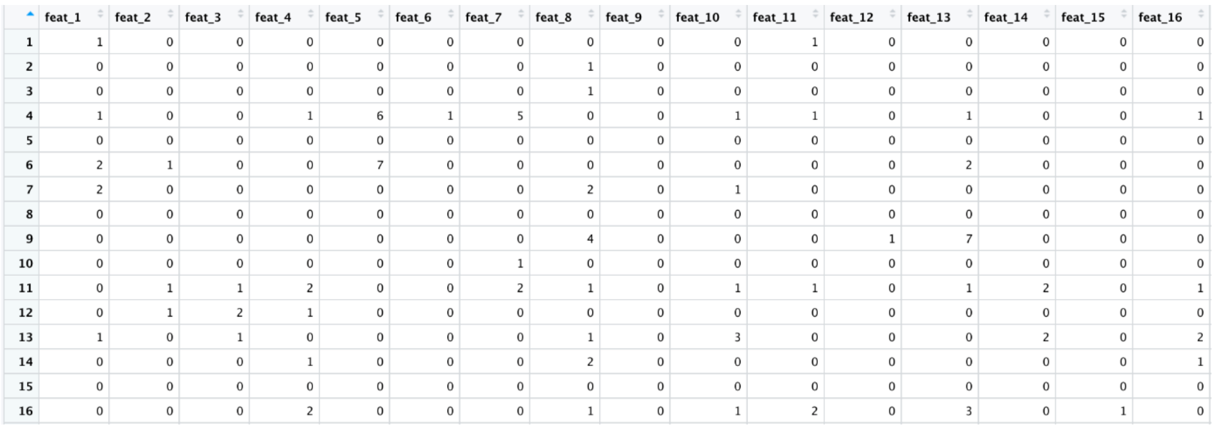
After getting accustomed with the data, the next section will provide a walkthrough of the necessary data preparation steps undertaken in order to enable accurate training of classifier models.

2.2.1 Data Preparation – Basics

To ensure that no data is missing as it might significantly influence the analysis at a later stage, the **colSums(is.na(data))** function is employed. For the Otto dataset, no missing values were identified.

The first column of the data includes the unique product ID which is not relevant for the classification as it does not help to sort a product into a category. Therefore, the entire column is removed.

```
# Removing first row (product ID is of no value to us)
data <- data[,-1]
```



	feat_1	feat_2	feat_3	feat_4	feat_5	feat_6	feat_7	feat_8	feat_9	feat_10	feat_11	feat_12	feat_13	feat_14	feat_15	feat_16
1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
4	1	0	0	1	6	1	5	0	0	1	1	0	1	0	0	1
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	2	1	0	0	7	0	0	0	0	0	0	0	2	0	0	0
7	2	0	0	0	0	0	0	2	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	4	0	0	0	1	7	0	0	0
10	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
11	0	1	1	2	0	0	2	1	0	1	1	0	1	2	0	1
12	0	1	2	1	0	0	0	0	0	0	0	0	0	0	0	0
13	1	0	1	0	0	0	0	1	0	3	0	0	0	2	0	2
14	0	0	0	1	0	0	0	2	0	0	0	0	0	0	0	1
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	2	0	0	0	1	0	1	2	0	3	0	1	0

Figure 3: Initial Dataset Excerpt

Although specific feature engineering is not possible in this case due to missing information regarding the nature and characteristics of each feature, it is necessary to transform the target variable into a factor to create classification models.

```r
# Setting target variable as factor
data$target <- as.factor(data$target)
```

Furthermore, the correlation between features is checked to avoid poor performance due to multicollinearity. The recommended threshold to exclude high correlation is 0.9. To be sure, the threshold was set to 0.85 and even with the lower value, no strong correlations were found.

```r
# Checking for high correlation, recommended threshold is 0.9
library(caret)
correlation.mat<-cor(data[,-94])
highcorrelation <- findCorrelation(correlation.mat, 0.85)
highcorrelation

## integer(0)
```

One method to reduce multicollinearity is the Principle Component Analysis (PCA) which basically extracts important features and creates new individual components. As mentioned before, in the case of Otto it was not necessary since no high correlation was detected. Besides, PCA was not implemented due to various reasons such as reduced accuracy and an increased complexity of the features. On one hand, a diminished number of features reduces the accuracy of the models as they do not build on the entire set. On the other hand, it simultaneously increases the complexity and impedes the analysis as it builds new components out of the features based on their correlation. The specification of the features in the Otto dataset are unknown and the newly formed components would therefore further complicate the analysis since the components are equally inconclusive. However, if PCA would have been used on this dataset, the features would have been reduced from 93 to around 76.

2.2.2 Data Preparation – Dataset Imbalance

The results of exploratory data analysis illuminated a substantial dataset as well as features with different scopes. In order to standardize the features and hence, the analysis, it is crucial to normalize the given data by using the **preprocess()** function integrated in the *caret* library.

```
# Normalization of variables/features
preProcValues <- preProcess(data, method = "range")
data <- predict(preProcValues, data)
summary(data)

##      feat_1             feat_2             feat_3            feat_4
##  Min.   :0.000000   Min.   :0.000000   Min.   :0.00000   Min.   :0.00000
##  1st Qu.:0.000000   1st Qu.:0.000000   1st Qu.:0.00000   1st Qu.:0.00000
##  Median :0.000000   Median :0.000000   Median :0.00000   Median :0.00000
##  Mean   :0.006339   Mean   :0.005158   Mean   :0.01409   Mean   :0.01113
##  3rd Qu.:0.000000   3rd Qu.:0.000000   3rd Qu.:0.00000   3rd Qu.:0.00000
##  Max.   :1.000000   Max.   :1.000000   Max.   :1.00000   Max.   :1.00000
```

In order to train the classification models and then test their accuracy it is necessary to create a sample which is then split into test and training data. Essentially, the training set is used to train the respective classification model to correctly categorize new products based on the given features. The test set is then utilized to test the model by exposing it to new data that it has not used before. Throughout this project a train test split of 80%-20% was used.

```
# Creating test and training data sets
set.seed(54321)
index <- createDataPartition(sample$target, p=0.8, list=FALSE)
training <- sample[index,]
test <- sample[-index,]
nrow(training)

## [1] 49507

nrow(test)

## [1] 12371
```

As illustrated earlier with the **table()** function, the observations are unequally distributed across the nine product classes.

For the purpose of balancing the categories and reducing the amount of data that is being analyzed to optimize computation time, downsampling is applied to the training set. This is also necessary since minority class observations in unbalanced datasets will not be effectively trained during the creation of a classifier. As a result, the classifier model would then perform poorly at predicting minority class observations when exposed to new data. As opposed to the initial presentation of this case study, a major mistake was identified and corrected.

In earlier versions of the code the whole data set was downsampled and then split into the test and training sets. This would imply that when a classifier is tested on the test sample, the test sample also consists of downsampled data. Subsequently the classifier is evaluated on data that is largely dissimilar to the data that would be fed into the model when new products are added to the Otto store.

```
# Applying downsampling to training set due to data set imbalance
library(caret)
set.seed(54321)
downsample <- downSample(training[, -ncol(training)], training$target, yname = "ta
rget")
summary(downsample$target)

## Class_1 Class_2 Class_3 Class_4 Class_5 Class_6 Class_7 Class_8 Class_9
##    1544    1544    1544    1544    1544    1544    1544    1544    1544


training<-downsample
```

At the end of the data preparation, the training and the test samples are summarized:

```
# Summary of finished test and training sets
summary(training$target)

## Class_1 Class_2 Class_3 Class_4 Class_5 Class_6 Class_7 Class_8 Class_9
##    1544    1544    1544    1544    1544    1544    1544    1544    1544


summary(test$target)

## Class_1 Class_2 Class_3 Class_4 Class_5 Class_6 Class_7 Class_8 Class_9
##     385    3224    1600     538     547    2827     567    1692     991
```

3 Classification

3.1 Report Setup and Cross Validation

Before the classification models are developed, a general setup is implemented to ensure consistency throughout the classification process. By implementing cross validation, the effectiveness and performance of predictive models can be measured. A data sample is divided into n folds, in this case a five-fold-cross-validation, whereas four training folds are used in fitting the model while the remaining one is used to validate its performance at the last instance. Every time a new fold is fitted and tested, the accuracy is captured, i.e. the average and the standard deviation of the model. In the end, the best hyperparameters are used to select the most accurate version of a model.

```r
# Setup of report and cross validation
TControl <- trainControl(method="cv", number=5)
report <- data.frame(Model=character(), Acc.Train=numeric(), Acc.Test=numeric())
```

The general format of the predictive model development is structured by running all taught classification models with the default values given in R. After running through each one, the approach with the most promising accuracy is further tuned and adjusted. Furthermore, at the beginning of every classification approach, the **set.seed()** function is implemented to ensure reproducibility since the same random numbers are generated when the code is run. The models are created by applying *caret's* **train()** function that can specify which type of classifier is to be created. This function sets up a grid of tuning parameters for a number of classification and regression routines, fits each model and calculates a resampling based performance measure. After training the classifier, the **predict()** function applies the model to the specified test sample. While there are multiple evaluation metrics available for comparing classifiers against each other, for the purposes of this report "Accuracy", as in the ratio of correct predictions compared to total input samples, was chosen to rank the models. In order to attain both the train and test accuracy of every classifier, a confusion matrix built from predictions and reference observations is created each time. These accuracies are then added to a report that will be examined at the end of this section. The costs of misclassification were largely omitted in the following models. While they do manifest themselves through the reduction of predictive accuracy to some degree, there was no additional weight applied. Since a misclassification will not have major negative consequences beyond customers potentially being shown nonsensical recommendations, this was deemed appropriate. This would not be the cases if misclassifications where more serious like it is the case in classifying creditworthiness for example.

3.2 K-Nearest Neighbor

The main principle behind the k-nearest neighbor model is that it allows classification based on the attributes of neighboring objects in the dataset. KNN does not explicitly build any model, it simply tags the new data entry-based learning from historical data.
The observations are classified by their proximity towards other labeled data points (i.e. observations). Depending on the calculated distance between the points, the frequency of the classes that are closest is evaluated. Then the new object is assigned to the most frequent class in its proximity. For this step, the normalization, introduced in the preparation part, is necessary to avoid weighting errors.

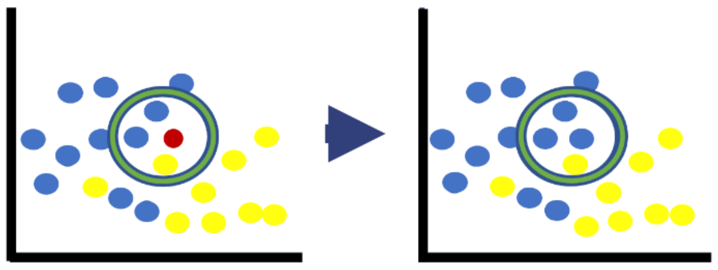


Figure 4: K-Nearest Neighbor Mechanism

```r
# K Nearest neighbors
set.seed(54321)
knnmodel <- train(target~., data=training, method="knn", trControl=TControl)
knnmodel
prediction.test <- predict(knnmodel, test[,-94],type="raw")
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
report <- rbind(report, data.frame(Model="k-NN", Acc.Train=0.00,
Acc.Test=accte$overall['Accuracy']))
report
```

KNN works well with small number of input variables but as the numbers of variables grow, the KNN algorithm struggles to predict the output of new data point. Since there are 93 predictor variable present in the dataset, the KNN classifier can be expected to perform comparatively poor.

3.3 Naïve Bayes Classifier

The Naïve Bayes classifier is based on the Bayes theorem, which describes the conditional probability of an outcome occurring based on prior knowledge of conditions or previous outcomes occurring.

$$P(B_i|A) = \frac{P(B_i) \cdot P(A|B_i)}{P(A)}$$

The performance of the Naïve Bayes classifier depends on the independence of the features. A big advantage of the method is that it requires only a small amount of training data, leading to shorter train times.

When using Naïve Bayes, it is important to make sure that no variable appears in the test data set that is not also present in the training data set. Otherwise, this would cause the probability to be 0 and therefore no prediction would be possible.
This is called *Zero Frequency* which is rather unlikely in the OTTO data set.
As mentioned before, the structure of the code, with the exception of the method within the **train()** function, is in line with the previous classification models. **Set.seed()** and the 5-fold cross validation set through *tControl* have not been changed. The Naïve Bayes Model requires a change of the method parameter to "nb".

```
# Naive Bayes Model
set.seed(54321)
nbmodel <- train(target~., data=training, method="nb", trControl=TControl)
nbmodel
prediction.train <- predict(nbmodel, training[,-94],type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']maybe
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
report <- rbind(report, data.frame(Model="Naive Bayes",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))
```

3.4 Decision Trees

The next classifications are three variations of the concept of *Decision Trees*. With these methods, datasets are analysed in a step-by-step analysis with a hierarchical structure which divides the data into subsets at every split as illustrated in Figure 5 below. Decision trees can therefore be seen as a map of different paths. The perfect decision tree can partition the training data until the final nodes contain only observations of a single class which is however not always achievable.

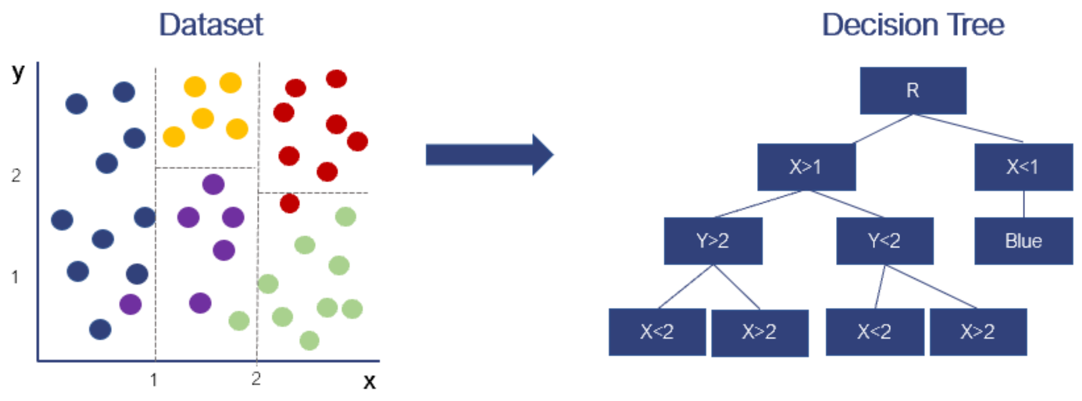


Figure 5: Decision Trees Mechanism

3.4.1 C5.0 Model

The C5.0 model makes decisions based on the split which provides the highest information gain; hence reducing entropy. As a result, the classification measure for this decision tree type is the information gain ratio. For each split the information gain is calculated by subtracting the weighted entropies of each branch from the original entropy.

In essence, the larger the information gain of a decision is, the more entropy is removed. Hence, the variance of the data is reduced and consequently the predictability of data improved.

The code for this model has the same structure as the previous models with the exception that the method is changed into *C5.0* for the **train()** function.

```
# C5.0 Model
set.seed(54321)
c5model <- train(target ~., data=training, method="C5.0", trControl=TControl)
c5model
prediction.train <- predict(c5model, training[,-94], type="raw")
prediction.test <- predict(c5model, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
report <- rbind(report, data.frame(Model="C5.0",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))
report
```

3.4.2 CART Model

In contrast to C5.0, the decisions of the CART model are based on the attribute with the highest information content. The classification measure here is the Gini-Index which evaluates the quality of splits, i.e. if they are better or worse than others in classifying data points correctly. Just like the three preceding classifications, the CART code is built the same except for the method, as previously explained.

```
# Cart Model
set.seed(54321)
cartmodel <- train(target ~., data=training, method="rpart", trControl=TControl)
cartmodel
prediction.train <- predict(cartmodel,training[,-94], type="raw")
prediction.test <- predict(cartmodel,test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
```

```r
report <- rbind(report, data.frame(Model="CART",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))
```

3.4.3 Random Forest

Random forest combines the simplicity of decision trees with flexibility resulting in a vast improvement in accuracy. As the name suggests this algorithm creates a forest of multiple decision trees opposed to a single tree like in CART or C5.0 model. For every tree a randomly chosen sample of datapoints is drawn. To classify a new object based on attributes each tree gives a classification. This event is called a vote. The forest chooses the classification having the most votes over all the other trees in the forest. In general, the more trees in the forest the more robust the prediction and thus higher accuracy. The accuracy of the model can be influenced by limiting the size and depth of the trees.

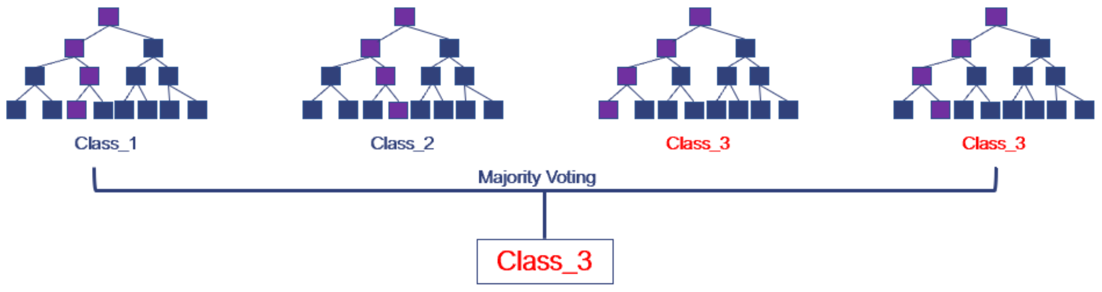


Figure 6: Random Forest - Voting Mechanism

```r
# Random forest
set.seed(54321)
rformodel <- train(target ~., data=training, method="rf", trControl=TControl)
rformodel
prediction.train <- predict(rformodel, training[,-94], type="raw")
prediction.test <- predict(rformodel, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
report <- rbind(report, data.frame(Model="Random Forest",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))
report
```

3.5 Neural Networks

Neural Networks are used for classification where an object can fall into one of at least two different categories. Thus, a neural network model was trained on the dataset to identify its accuracy. Neural networks analyse datasets by sending data randomly through a multiple layered network. The aim is to obtain high accuracy in prediction. Meaning the neural network ideally should predict a value that is as close to the actual output as possible. An improvement in accuracy can be achieved through continuous training. The model finetunes itself by analyzing new data.

For this project a limited dataset was analyzed and in consequence training was also limited. In reality, at Otto Group new data is collected daily. It is therefore likely that the Otto group would see gains in accuracy when more training data is provided. To train the neural network, the method parameter is change to “nnet”.

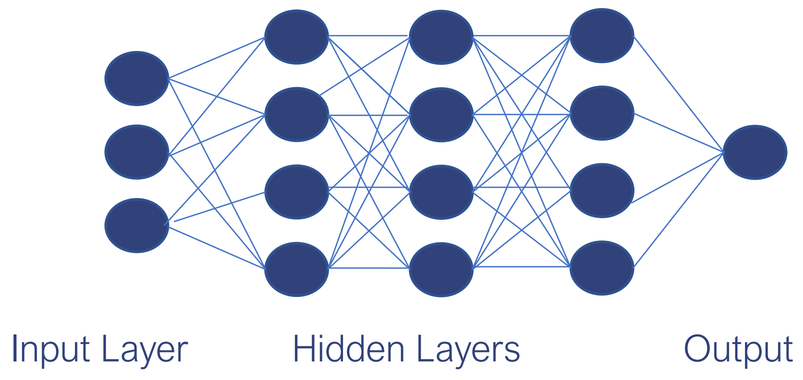


Figure 7: Neural Network - Layer Structure

```
# Neural Network Model
set.seed(54321)
nnmodel <- train(target ~., data=training, method="nnet", trControl=TControl)
nnmodel
prediction.train <- predict(nnmodel, training[,-94], type="raw")
prediction.test <- predict(nnmodel, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
report <- rbind(report, data.frame(Model="Neural Network",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))
```

3.6 Support Vector Machines Linear and Radial

As a next step, linear and radial Support Vector Machine models were trained to evaluate their overall suitability compared to the previous methods.

With the support vector machine algorithm, it is possible to find a hyperplane in an N-dimensional space that classifies the data points with N reflecting the number of features. There are a lot of possible hyperplanes that could be chosen to separate the data points. Therefore, the objective is to find a plane that has the maximum margin. Maximizing the margin distance provides some amplification so that future data points can be classified with greater certainty. However, the margin needs to be chosen according to the perceived misclassification costs. As discussed in the beginning, Otto does not face high misclassification costs if a product is classified incorrectly. A rather small C value (penalty parameter) should therefore not impede the predictive model as much as in other cases.

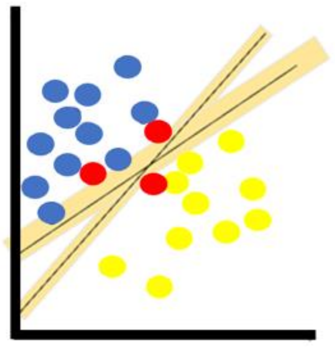


Figure 8: Support Vector Machine Model – Illustration (2 Dimensional)

To run the Linear Support Vector Machine model, the code used with the previous classifications is simply altered by changing the method to “svmLinear”:

```
# Support Vector Machine model
set.seed(54321)
svmmodel.l <- train(target ~., data=training, method="svmLinear",
trControl=TControl)
svmmodel.l
prediction.train <- predict(svmmodel.l, training[,-94], type="raw")
prediction.test <- predict(svmmodel.l, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
report <- rbind(report, data.frame(Model="SVM (Linear)",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))
```

The Radial Support Vector Machine model then requires a change of the method to “svmRadial”.

```
# Radial Support Vector Machine Model
set.seed(54321)
svmmodel.r <- train(target ~., data=training, method="svmRadial",
trControl=TControl)
svmmodel.r
prediction.train <- predict(svmmodel.r, training[,-94], type="raw")
prediction.test <- predict(svmmodel.r, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
```

```r
report <- rbind(report, data.frame(Model="SVM (Radial)",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))
```

4 Results

4.1 Comparison of Model Accuracies

After having trained 8 different classifiers and storing their respective test and train accuracies, calling the report delivers the following output:

```
Report

Accuracy                  Model  Acc.Train    Acc.Test

## Accuracy                k-NN      0.0000000    0.6847466

## Accuracy1        Naive Bayes      0.5615285    0.6847466

## Accuracy2               C5.0      0.9863990    0.7494140

## Accuracy3               CART      0.2633851    0.1779161

## Accuracy4      Random Forest      1.0000000    0.7566890

## Accuracy5     Neural Network      0.7301382    0.6937192

## Accuracy6       SVM (Linear)      0.7700058    0.7217687

## Accuracy7       SVM (Radial)      0.8288716    0.7271037
```

The report shows that out of all the different classifiers the random forest model delivers the highest test accuracy. Since the goal is to correctly classify new products, this model is deemed as the best one. However, the random forest’s train accuracy of 100%, leading to a train test accuracy discrepancy of roughly 25%, is concerning. A discrepancy between the two accuracies of this size indicates a risk of overfitting which is to be avoided.

Overfitting happens when the model has too much freedom to fit the data. The more complex a model is, the more likely it is to overfit. Some possible solutions to the problem of overfitting include applying cross-validation and reducing model complexity. The report also shows that the support vector machine trained on a radial kernel performs good as well. While its test accuracy is lower than that of the random forest model, its lower train accuracy implies less risk of overfitting.

5 Tuning

After training and comparing the classifiers on *caret’s* default hyperparameters, tuning can be applied to improve the characteristics of some of the classifiers. This can be done by changing the hyperparameters available and checking which combination leads to the optimal train and test accuracies. Since tuning can take significant amounts of time, only the two most promising classifiers where selected for further tuning. Since tuning requires the retraining of the classifier for every hyperparameter and possibly even every combination of hyperparameters specified computing times can quickly become unpractical. In order to speed up processing times, the "*doParallel*" library was used as it allows more efficient multicore processing.

5.1 Tuning Radial Support Vector Machine

Tuning radial support vector machines is mainly done by adjusting the models C and sigma values. As previously explained, the Support Vector Machine model is exploring hyperplanes that separate the dataset best in its target classes. However, this doesn't always mean that a perfect separation is desirable. In some cases, hyperplanes need to be set in regard to potential datapoints that overlap the initial boundary of the classes to ensure the majority of new data can be classified correctly. The soft margin constant C determines how "strict" the margin around the boundaries, i.e. hyperplanes, is set. Hence, the smaller the C value is, the bigger the margin that allows to ignore some misclassified datapoints. If a perfect separation is needed, a high C value has to be selected. On the other hand, sigma regulates the flexibility of the hyperplanes. Therefore, with small sigma values, the flexibility of the boundaries increases, and the data can be fitted more accurately. Nonetheless, small sigma values also increase the likelihood of overfitting.

Trying for different combinations of these values can be achieved by including them into a tune grid for the model as was done in the following code:

```
library(doParallel)
library(caret)
cores <- makeCluster(detectCores()-1)
registerDoParallel(cores = cores)
svmgrid <- expand.grid(sigma = c(0,0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9),
                       C = c(0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2))
set.seed(54321)
tunedsvm.r <- train(target ~., data=training, method="svmRadial",
                    trControl=TControl, tuneGrid = svmgrid)
tuningresultssvm.r <- data.frame(tunedsvm.r[["results"]])
```

Applying the *ggplot* library allows us to visualize the results:

```
# Plotting Accuracies across multiple parameter combinations
ggplot(tuningresultssvm.r, aes(x = C, y = Accuracy, color= as.factor(sigma))) +
  geom_point()+
  scale_x_continuous(name="C parameter",) +
  scale_y_continuous(name="Accuracy",) +
  ggtitle("Accuracy Across Multiple Values for C and Sigma")+
  theme_bw()
```

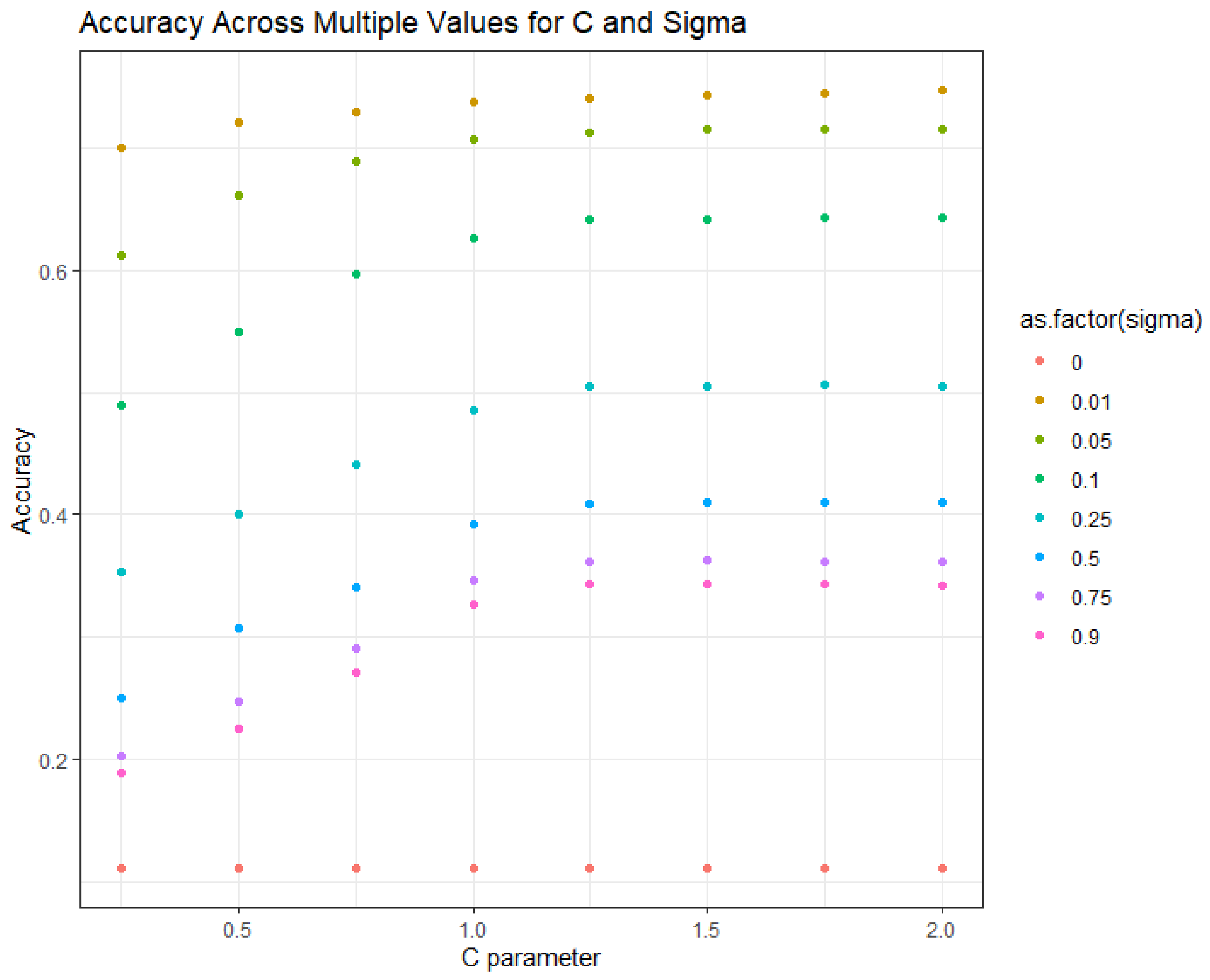


Figure 9: SVM - Accuracy Tuning via C, Sigma value adjustment

As can be seen in the plot above, different values for sigma have a sizeable effect on train accuracy. C values for a Sigma of 0.01 are the most proficient in predicting across the board. Checking the $bestTune variable of the support vector machine shows that the highest train accuracy attained (74,72%) was with the values of sigma=0.01 and C =2.

```
# Print the best tuning parameter sigma and C that maximizes model accuracy
tunedsvm.r(accuracy)
tunedsvm.r$bestTune
##    sigma C
## 16  0.01 2
stopCluster(cores)
```

The accuracy of the tuned radial support vector machine was shown to be worse than the one of the support vector machines trained on *caret's* default values. While more focused tuning, likely by checking incremental values around a sigma of 0.01, would have verry likely allowed increases in train accuracy towards the vicinity of the accuracy attained by the default values, the initial tuning shows to be ineffective. Instead of investing further resources into more sophisticated tuning of the support vector machine, it was decided to move on to tuning the random forest classifier.

5.2 Tuning Random Forest

While the random forest model trained on the default values selected by the *caret* package deliver descent results, it should be possible to improve upon them even further through tuning of the model. By focusing on finding the optimal hyperparameters that are available for random forest, incremental improvements in the model can be obtained. When considering the size of the data the model is applied to, accuracy gains of a few precent will already be of value. While there are numerous different hyperparameters available, the parameters “ntree”, “.mtry” and “maxnodes” are arguably the most important ones in the context of tuning random forests. Besides increasing accuracy, tuning also allows addressing the problem of overfitting. As can be seen from the preliminary report comparing the different classification models, the random forest model sports a significant gap between its respective train and test accuracy (around 25%). Since the classification method that is ultimately chosen will have to deal with a constant stream of new data for products that need to be classified, the random forest model can be expected to perform significantly worse in predicting new products when compared to the ones already present in the data. Therefore, it will be necessary to narrow the 25% gap in train and test accuracy to a manageable level. Luckily this can also be achieved through tuning the model. What follows is a brief description of the hyperparameters chosen and the approach taken for tuning them.

ntree: This parameter describes how many different trees are generated when creating the tree ensemble that is random forest. Usually, a higher ntree value will increase a model’s accuracy since there are more trees available when voting occurs during classification. However, more trees can also have a slight impact on overfitting the model. More importantly, ntree is also the parameter that has the single highest impact on the computational power required to run the random forest model. While it took roughly 25 minutes to train an individual random forest model, tuning across multiple hyperparameters with high values for ntree led to a seemingly exponential increase in computing time necessary. So much so that it was not possible to perform the tuning operations that were planned. It was therefore decided to cap the maximum value for ntree at 150. Besides limiting computing time, this led to a slight decrease in the overfitting gap of the model. Through trial and error, it was also gathered that the decrease in predictive accuracy caused by setting a comparatively low value for ntree was relatively small and within acceptable margins.

.mtry: This parameter decides how many different variables are considered at each node of the decision trees that are trained during the creation of a random forest model. *Caret* sets the default at the square root of the amount of predictor variables available. In this case this is sqrt(93). While this default value is usually a good starting point, it is not guaranteed that it is the optimal value and can therefore be tuned to attain higher accuracies.

Maxnodes: The maxnodes parameter limits the number of decision nodes that are contained within the decision trees trained. Lower values of maxnodes will therefore cause trees to be cut off earlier than they would be if they were fully trained. While lowering the value for maxnodes will decrease a model's train and test accuracy, removing tree complexity by lowering its maxnodes can prove to be an effective way of addressing the problem of overfitting.

5.2.1 Tuning Random Forest – ntree & .mtry

The initial approach undertaken was to first tune for an optimal .mtry parameter and then to tune for ntree to gain maximum accuracy, before then reducing overfitting by tuning maxnodes. Tuning values for .mtry is relatively straight forward, and can be achieved by creating a tune grid containing the different values for .mtry that are to be considered. Including the tunegrid into *caret*'*s* train function will then lead to a new random forest being trained for every single mtry value included in the grid. While this approach can already be sufficient for the goal of increasing model accuracy, it omits the dynamic effects between hyperparameters. The different hyper parameters of random forest have shown to exhibit interaction effects between them. For example, the optimal .mtry value for a model trained on ntree(100) may differ from the one of a model trained on ntree(200). In order to study these interaction effects with the added benefit of hopefully identifying the optimal parameter combination, a different approach must be chosen. By applying the **tunegrid()** function to a series of different models trained on different values for ntree, the interaction between the two parameters can be studied more closely. As mentioned above, it was decided to cap the maximum value for ntree at 150. For ntree values below 150, lower values were chosen in increments of 25. Unfortunately, it is not possible to include the different ntree values in the tune grid as well since they are directly passed on to the train function, requiring the coding of numerous different models. An elegant approach to this problem would include running a loop, cycling through the different ntree values. However, in this case constructing and testing such a loop proved to take more time than just executing the code manually for the different values, leading to the code seen below:

```
# Tuning of Random Forest ----------------------------------------
# Tuning across multiple parameters requires allot of computation. The
"doparallel" libary allows for better multicore processing, speeding up computing
times
library(doParallel)
library(caret)
cores <- makeCluster(detectCores()-1)
registerDoParallel(cores = cores)

# Test for optimal .mtry and ntree parameter
set.seed(54321)
mtrygrid <- expand.grid(mtry = c(3:20))
rformodelmtryn25 <- train(target ~., data=training, method="rf",metric =
"Accuracy", tuneGrid=mtrygrid, trControl = TControl, importance = TRUE,ntree=25)
rformodelmtryn50 <- train(target ~., data=training, method="rf",metric =
"Accuracy", tuneGrid=mtrygrid, trControl = TControl, importance = TRUE,ntree=50)
rformodelmtryn75 <- train(target ~., data=training, method="rf",metric =
"Accuracy", tuneGrid=mtrygrid, trControl = TControl, importance = TRUE,ntree=75)
rformodelmtryn100 <- train(target ~., data=training, method="rf",metric =
"Accuracy", tuneGrid=mtrygrid, trControl = TControl, importance = TRUE,ntree=100)
rformodelmtryn125 <- train(target ~., data=training, method="rf",metric =
"Accuracy", tuneGrid=mtrygrid, trControl = TControl, importance = TRUE,ntree=125)
rformodelmtryn150 <- train(target ~., data=training, method="rf",metric =
"Accuracy", tuneGrid=mtrygrid, trControl = TControl, importance = TRUE,ntree=150)


# Storing the results in dataframes for plotting and adding a Variable to indicate
source model
tuningresultsmtryn25 <- data.frame(rformodelmtryn25[["results"]])
tuningresultsmtryn25$ntree <- 'Ntree 25'
tuningresultsmtryn50 <- data.frame(rformodelmtryn50[["results"]])
tuningresultsmtryn50$ntree <- 'Ntree 50'
tuningresultsmtryn75 <- data.frame(rformodelmtryn75[["results"]])
tuningresultsmtryn75$ntree <- 'Ntree 75'
tuningresultsmtryn100 <- data.frame(rformodelmtryn100[["results"]])
tuningresultsmtryn100$ntree <- 'Ntree 100'
tuningresultsmtryn125 <- data.frame(rformodelmtryn125[["results"]])
tuningresultsmtryn125$ntree <- 'Ntree 125'
tuningresultsmtryn150 <- data.frame(rformodelmtryn150[["results"]])
tuningresultsmtryn150$ntree <- 'Ntree 150'
plotdata <- rbind.data.frame(tuningresultsmtryn25, tuningresultsmtryn50,
tuningresultsmtryn75, tuningresultsmtryn100, tuningresultsmtryn125,
tuningresultsmtryn150)
stopCluster(cores)
```

After storing the results from the code above into various data frames, the *ggplot* library can be applied to visualize the results.

```
# Plotting model accuracies across different Values for .mtry and ntree
library(caret)
ggplot(plotdata, aes(x = mtry, y = Accuracy))+
  geom_line(aes(colour = ntree), size = 1.5)+
  scale_x_continuous(name="Mtry Parameter",) +
  scale_y_continuous(name="Accuracy",) +
  ggtitle("Accuracy Across Multiple Values for Mtry and Ntree")+
  theme_bw()
```

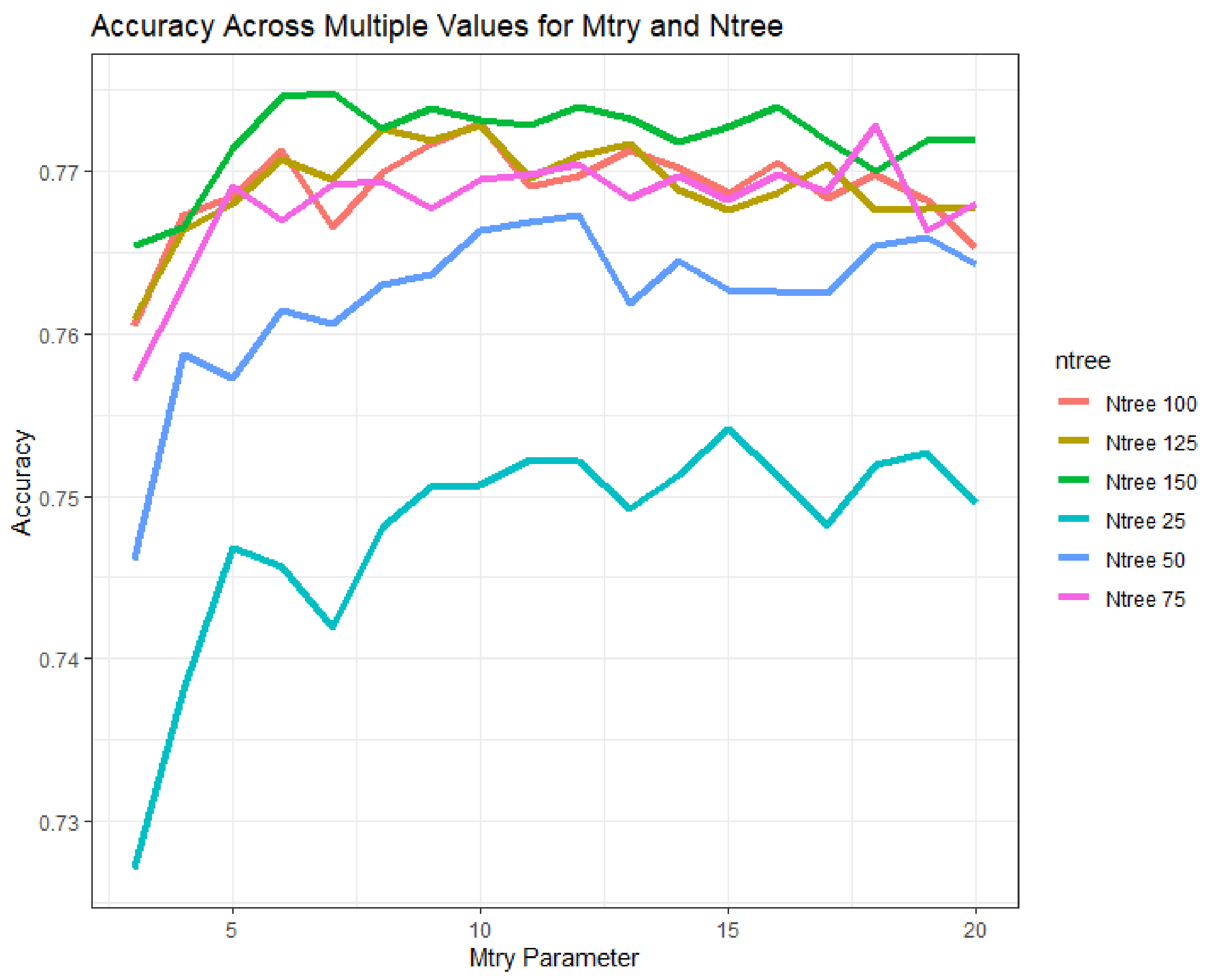


Figure 10: Random Forest - Accuracy Tuning - .mtry, ntree

```
# best .mtry = 7 .ntree = 150
```

This plot shows a good representation of the interaction effects between the .mtry and ntree parameter. While ntree values of 25 and 50 lead to a sizeable drop in train accuracy, different ntree values from 75 upwards show little effect on accuracy. The optimal parameter combination in this limited testing is found to be at a .mtry value of 7 and ntree value of 150. Furthermore, the small difference in accuracy across the board for ntree values above 75 seem to support the constraint of capping at ntree at 150 since higher values will likely have minimal effects on model accuracy.

5.2.2 Tuning Maxnodes

After finding the optimal values for .mtry and ntree, the problem of overfitting still persist. As mentioned before, tuning maxnodes will allow alleviating the problem somewhat. While it is likely that the maxnodes parameter also shows some interaction effect between ntree and .mtry, testing across three parameter combinations would have exceeded the resources and scope of this project. It would have also required three-dimensional visualization in *ggplot*. Doable but time intensive as well.

After setting the optimal .mtry and ntree variables in *caret's* train function, multiple models with differing values in maxnodes where trained. Since the primary focus of this tuning is to address overfitting, it was not possible to simply work with a tunegrid. Instead the model had to be trained and applied with a full confusion matrix each time in order to attain both the train and test accuracy. This was done in the following way:

```
# Maxnode Tuning ----------------------------------------------------------

#Different Values for maxnodes parameter to reduce overfitting with mtry and ntree
held constant

library(caret)
tunedrfmn500 <- train(target ~., data=training, method="rf", trControl=TControl,
.mtry=7, ntree=150,maxnodes=500)
prediction.train <- predict(tunedrfmn500, training[,-94], type="raw")
prediction.test <- predict(tunedrfmn500, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
Maxnodereport <- rbind(Maxnodereport, data.frame(Model="tunedrfmn500",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))

Maxnodereport <- data.frame(Model=character(), Acc.Train=numeric(),
Acc.Test=numeric())
tunedrfmn750 <- train(target ~., data=training, method="rf", trControl=TControl,
.mtry=7, ntree=150,maxnodes=750)
prediction.train <- predict(tunedrfmn750, training[,-94], type="raw")
prediction.test <- predict(tunedrfmn750, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
Maxnodereport <- rbind(Maxnodereport, data.frame(Model="tunedrfmn750",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))

tunedrfmn1000 <- train(target ~., data=training, method="rf", trControl=TControl,
.mtry=7, ntree=150,maxnodes=1000)
prediction.train <- predict(tunedrfmn1000, training[,-94], type="raw")
prediction.test <- predict(tunedrfmn1000, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
Maxnodereport <- rbind(Maxnodereport, data.frame(Model="tunedrfmn1000",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))

tunedrfmn1250 <- train(target ~., data=training, method="rf", trControl=TControl,
.mtry=7, ntree=150,maxnodes=1250)
prediction.train <- predict(tunedrfmn1250, training[,-94], type="raw")
prediction.test <- predict(tunedrfmn1250, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
Maxnodereport <- rbind(Maxnodereport, data.frame(Model="tunedrfmn1250",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))

tunedrfmn1500 <- train(target ~., data=training, method="rf", trControl=TControl,
.mtry=7, ntree=150,maxnodes=1500)
prediction.train <- predict(tunedrfmn1500, training[,-94], type="raw")
prediction.test <- predict(tunedrfmn1500, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
Maxnodereport <- rbind(Maxnodereport, data.frame(Model="tunedrfmn1500",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))

tunedrfmn2000 <- train(target ~., data=training, method="rf", trControl=TControl,
.mtry=7, ntree=150,maxnodes=2000)
prediction.train <- predict(tunedrfmn2000, training[,-94], type="raw")
prediction.test <- predict(tunedrfmn2000, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
Maxnodereport <- rbind(Maxnodereport, data.frame(Model="tunedrfmn2000",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))

Maxnodereport$maxnodes <- c(500,750,1000,1250,1500,2000)
Maxnodereport
```

```
                  Model Acc.Train  Acc.Test maxnodes

Accuracy   tunedrfmn500 0.7615861 0.6609005      500

Accuracy1  tunedrfmn750 0.8221071 0.6943659      750

Accuracy2 tunedrfmn1000 0.8702504 0.7085927     1000

Accuracy3 tunedrfmn1250 0.9074554 0.7204753     1250

Accuracy4 tunedrfmn1500 0.9379678 0.7321963     1500

Accuracy5 tunedrfmn2000 0.9832326 0.7425430     2000
```

To visualize the results the following *ggplot* function was applied.

```
#plotting results
ggplot(Maxnodereport, aes(maxnodes))+
  geom_line(aes(y=Acc.Train, color = "Acc.Train"), size = 1.5)+
  geom_line(aes(y= Acc.Test, color = "ACC.Test"), size = 1.5)+
  scale_x_continuous(name="maxnodes",) +
  scale_y_continuous(name="Accuracy",) +
  ggtitle("Accuracy Across Multiple Values for Mtry and Ntree")+
  theme_bw()
```

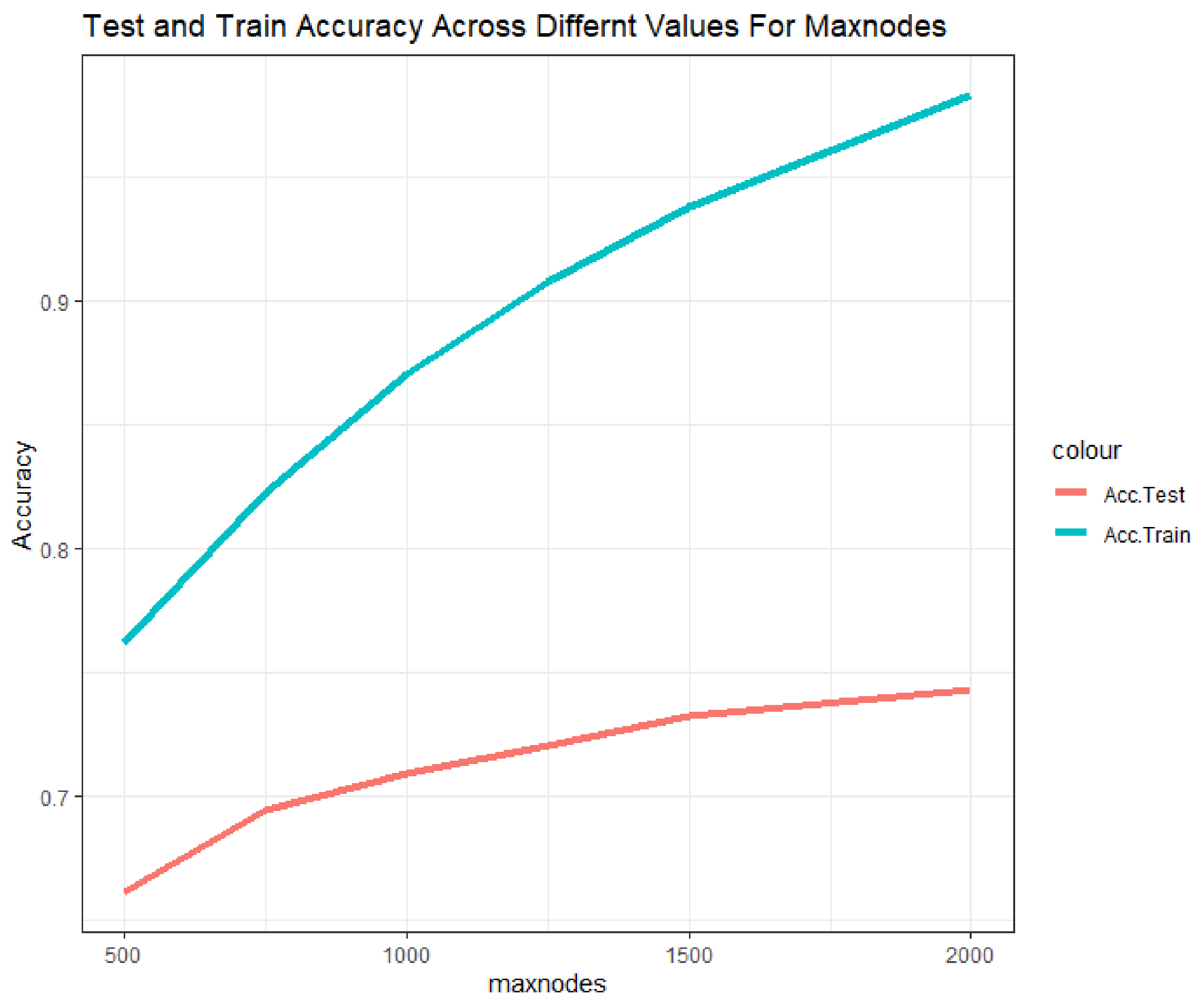


Figure 11: Random Forest - Test and Training Accuracy Development - Maxnodes Tuning

As can be seen in the plot above reducing the maxnodes parameter reduces both the models test and train accuracy. Training the model on maxnodes 2000 produces accuracies very similar to the ones observed in the initial report for models without tuning (Acc.Train= 0.98/Acc.Test = 0.74). The plot also shows that reducing maxnodes decreases train accuracy at a faster rate than the corresponding training accuracy, thereby addressing the problem of overfitting to some degree. At a value of maxnodes 500 the difference between the training and test accuracy (Acc.Train= 0.76/Acc.Test = 0.66) is decreased to roughly 10 percentage points.

At this point it was decided that this discrepancy is acceptable and not worth further sacrificing test accuracy for. Therefore, the final value for maxnodes was chosen to be 500.

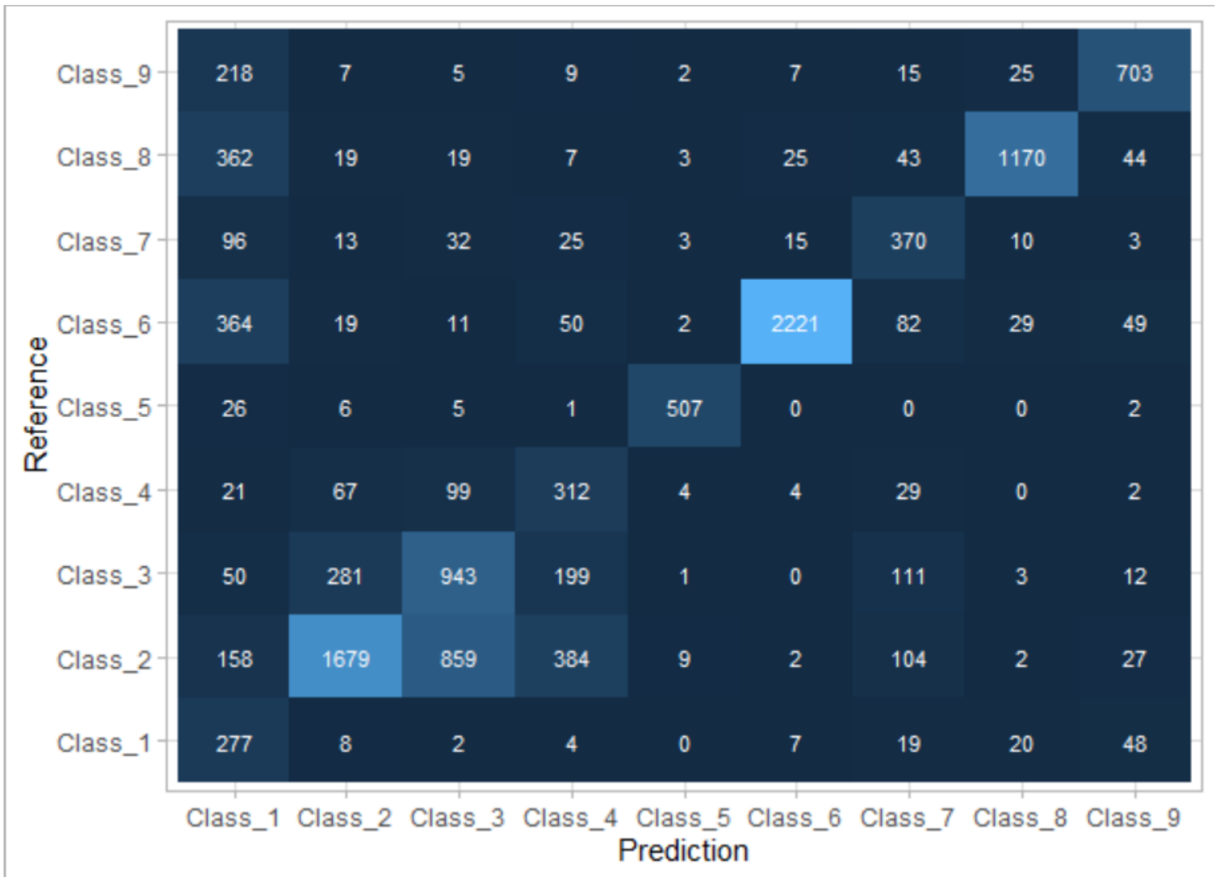
5.3 Final Random Forest

For the final random forest model the parameters of ntree = 150 .mtry = 7 and maxnodes = 500 where chosen. While a test accuracy of roughly 66% is not ideal, losing accuracy had to be tolerated to address overfitting. Rerunning the code for the final model allows for a closer look at how the model classifies observations into the respective classes:

```
# rerunning optimal model to attain confusion matrix and adding accuracy to report
tunedrfmn500 <- train(target ~., data=training, method="rf", trControl=TControl,
.mtry=7, ntree=150,maxnodes=500)
prediction.train <- predict(tunedrfmn500, training[,-94], type="raw")
prediction.test <- predict(tunedrfmn500, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, training[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
report <- rbind(report, data.frame(Model="Tuned Random Forrest",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))
accte$table
##           Reference
## Prediction Class_1 Class_2 Class_3 Class_4 Class_5 Class_6 Class_7 Class_8 Class_9
##    Class_1     277     158      50      21      26     364      96     362     218
##    Class_2       8    1679     281      67       6      19      13      19       7
##    Class_3       2     859     943      99       5      11      32      19       5
##    Class_4       4     384     199     312       1      50      25       7       9
##    Class_5       0       9       1       4     507       2       3       3       2
##    Class_6       7       2       0       4       0    2221      15      25       7
##    Class_7      19     104     111      29       0      82     370      43      15
##    Class_8      20       2       3       0       0      29      10    1170      25
##    Class_9      48      27      12       2       2      49       3      44     703
```

Applying *ggplot* allows us to visualize the confusion matrix.

```
library(ggplot2)
cm_d <- as.data.frame(accte$table)
ggplot(data = cm_d, aes(x = Prediction , y =  Reference, fill = Freq))+
  geom_tile() +
  geom_text(aes(label = paste(Freq)), color = 'white', size = 3) +
  theme_light() +
  guides(fill=FALSE)
```



Reference \ Prediction	Class_1	Class_2	Class_3	Class_4	Class_5	Class_6	Class_7	Class_8	Class_9
Class_9	218	7	5	9	2	7	15	25	703
Class_8	362	19	19	7	3	25	43	1170	44
Class_7	96	13	32	25	3	15	370	10	3
Class_6	364	19	11	50	2	2221	82	29	49
Class_5	26	6	5	1	507	0	0	0	2
Class_4	21	67	99	312	4	4	29	0	2
Class_3	50	281	943	199	1	0	111	3	12
Class_2	158	1679	859	384	9	2	104	2	27
Class_1	277	8	2	4	0	7	19	20	48

Figure 12: Tuned Random Forest - Confusion Matrix - Observation Distribution

From the confusion matrix we can observe that the tuned random forest model has difficulties differentiating between products belonging to class 2,3 and 4. Were this model to be applied in a real world setting, the otto group should consider having an employee focus efforts on manually checking and correcting classifications of products belonging to class 2,3 and 4 since the majority of misclassifications occur in this area.

6 Revisiting Sampling Training Data

Up until now all models have been trained on a downsampled training set. While downsampling is an effective method for balancing data sets, it also implies a loss of information when majority class observations are removed. This information may be beneficial for training the classifiers, which is why applying alternative sampling methods may be worthwhile. Instead of simply downsampling there are a few other methods of sampling available that were considered:

Upsampling: instead of removing majority class observations, minority class observations are synthesized until all classes are at the level of the original majority class. While up sampling may be of use here, it was found to increase the already large dataset to a point where computation times became unmanageable, making it hard to tune models. Upsampling was ultimately not pursued.

```
#applying Upsampling
trainingUP <- sample[index,]
trainingUP <- upSample(trainingUP[, -ncol(trainingUP)], trainingUP$target, yname =
"target")
nrow(trainingUP)
## [1] 116082
```

Over-undersampling: describes a hybrid method of applying both over and under sampling in one function. Combining the benefits and drawbacks of both methods. However, the **ov.un sample()** function in *caret* is not applicable for multiclass datasets and could therefore not be used. While it might be possible to manually apply over and under sampling to the individual classes (or to identifying a different package that allows over under sampling of multiclass data), other hybrid sampling methods were found to be more straight forward in their application.

ROSE: the "Random Over-Sampling Examples” function creates synthetic balanced samples promising to improve classifier accuracy. This is done by synthesizing examples from conditional density estimates of the target classes. However, similar to the **ov.un** function, it is only applicable to binary classification problems and was therefore not suitable for the dataset under consideration.

SMOTE: Synthetic minority oversampling is hybrid resampling technique that can be applied to multiclass data. The SMOTE algorithm generates minority class observations by using nearest neighbor methods. At the same time majority class observations are downsampled to balance the dataset. It was therefore chosen for a final comparison against the models trained strictly on downsampled data.

```
# Applying other resampling Methods ---------------------------------------
#applying Synthetic Minority Over-sampling Technique
library(DMwR)
trainingSMOTEs <- sample[index,]
trainingSMOTE <- SMOTE(target ~ ., data  = trainingSMOTEs, perc.over=200,
perc.under=400)
# check for missing values potentially created through SMOTE
colSums(is.na(trainingSMOTE))
## 0
#compare different training sets
nrow(training)
## [1] 13896
nrow(trainingUP)
## [1] 116082
nrow(trainingSMOTE)
## [1] 16984
```

After creating the new training set, a new random forest classifier is created using this new set. To make this comparison fair, the ntree parameter was held constant at 150 as it was the case in prior models. Since the new training set is largely different from the downsampled one, it may be the case that the optimal .mtry parameter used earlier has changed as well. Instead of redoing the entire tuning process for the prior random forest, the hyper parameter tuneLength = 15 was added to the train function. This prompts *caret* to try 15 different .mtry values at random. While it may not find the optimal parameter, it should identify one that is sufficiently close for our purposes.

```
# Random forest trained on SMOTE data
library(caret)
set.seed(54321)
rformodelSMOTE <- train(target ~., data=trainingSMOTE, method="rf",
trControl=TControl,importance = TRUE, ntree=150, tuneLength  = 15)
plot(rformodelSMOTE)
```

Applying the *ggplot* library allows the visualization of accuracies across the randomly selected values for .mtry.

```
ggplot(rformodelSMOTE, aes(x = mtry, y = Accuracy))+
  geom_line()+
  scale_x_continuous(name="Mtry Parameter",) +
  scale_y_continuous(name="Accuracy",) +
  ggtitle("Rfor Model accuracy across .mtry values and Trained on SMOTE Data")+
  theme_bw()
```

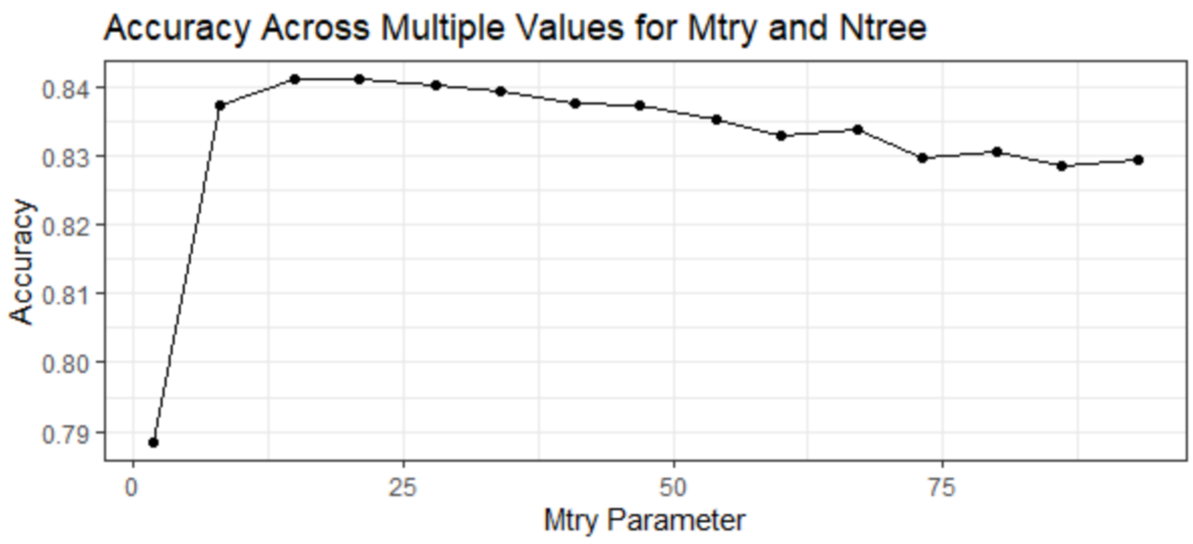


Figure 13: Random Forest - Tuning Accuracy - Mtry

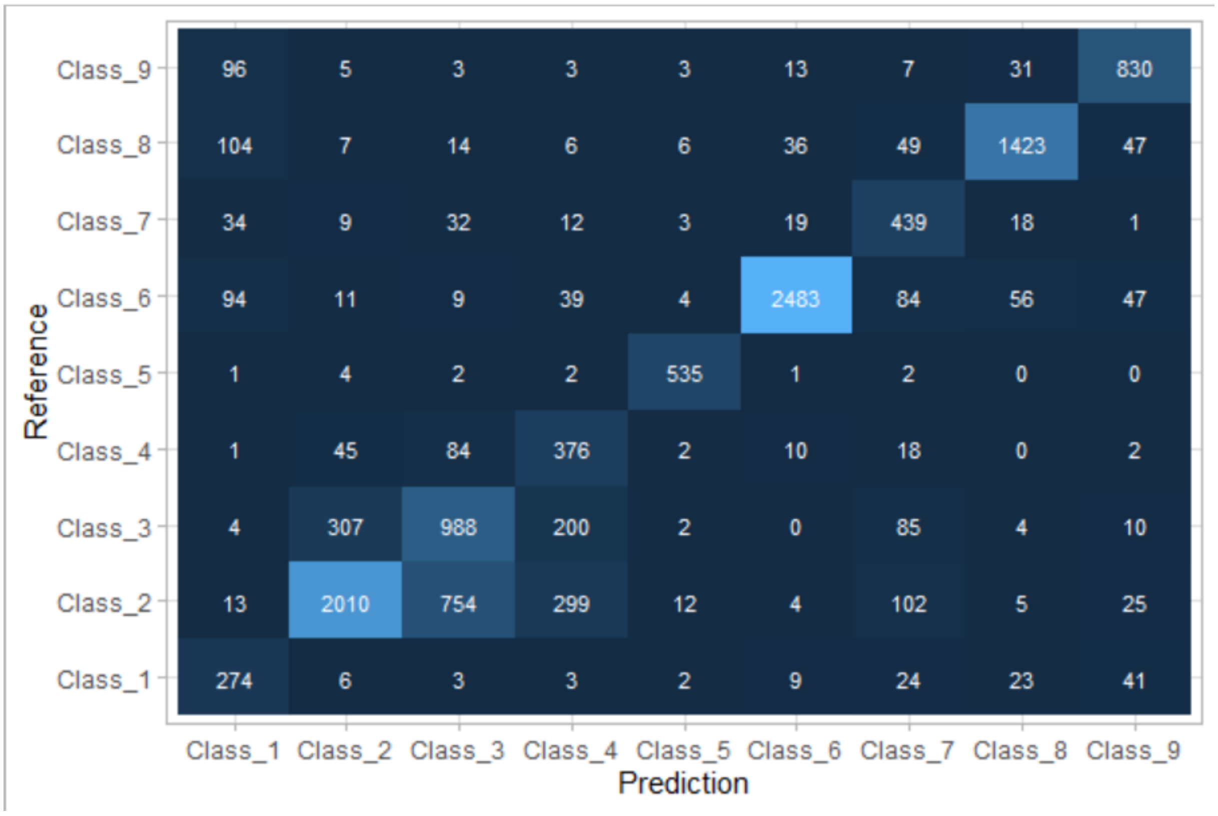
As expected the optimal .mtry parameter has changed and will from now on be held constant at 21. After identifying the best. mtry value of the ones trained for, the model is applied to the test data and a corresponding confusion matrix is generated.

```r
set.seed(54321)
rformodelSMOTE <- train(target ~., data=trainingSMOTE, method="rf",
trControl=TControl,importance = TRUE, ntree=150, .mtry=21)
prediction.train <- predict(rformodel, trainingSMOTE[,-94], type="raw")
prediction.test <- predict(rformodel, test[,-94], type="raw")
acctr <- confusionMatrix(prediction.train, trainingSMOTE[,94])
acctr$overall['Accuracy']
accte <- confusionMatrix(prediction.test, test[,94])
accte$overall['Accuracy']
accte$table
report <- rbind(report, data.frame(Model="Tuned Random Forest SMOTE",
Acc.Train=acctr$overall['Accuracy'], Acc.Test=accte$overall['Accuracy']))
```

```r
report
##                                  Model Acc.Train  Acc.Test
## Accuracy                          k-NN 0.0000000 0.6847466
## Accuracy1                  Naive Bayes 0.5615285 0.6847466
## Accuracy2                         C5.0 0.9863990 0.7494140
## Accuracy3                         CART 0.2633851 0.1779161
## Accuracy4                Random Forest 1.0000000 0.7566890
## Accuracy5               Neural Network 0.7301382 0.6937192
## Accuracy6                 SVM (Linear) 0.7700058 0.7217687
## Accuracy7                 SVM (Radial) 0.8288716 0.7271037
## Accuracy8        Tuned Random Forrest 0.7629534 0.6588796
## Accuracy9  Tuned Random Forest SMOTE 0.8573363 0.7564465
```

From this report we can gather, that the random forest trained on SMOTE data performs better than the tuned random forest model with downsampled data. While the test accuracy of the "Tuned random forest SMOTE" is very similar to the one obtained by the initial untuned random forest. The lower train accuracy shows a smaller risk of overfitting. The discrepancy in this case is roughly 10%, a threshold we have deemed as acceptable during the maxnode tuning of the random forest model trained on downsampled data. Therefore, it is not necessary to reduce tree complexity by decreasing maxnodes, allowing us to obtain a final test accuracy of 75,64%. From the many classifiers that were trained throughout this report a random forest Model trained on SMOTE data with the parameters of ntree 150 and .mtry 21 held constant is the one that is optimal. For a final step a new confusion matrix is generated to check whether the distribution of misclassification has changed from the one of the random forest trained on downsampled data.

```r
library(ggplot2)
cm_d <- as.data.frame(accte$table)
ggplot(data = cm_d, aes(x = Prediction , y =  Reference, fill = Freq))+
  geom_tile() +
  geom_text(aes(label = paste(Freq)), color = 'white', size = 3) +
  theme_light() +
  guides(fill=FALSE)
```



Reference \ Prediction	Class_1	Class_2	Class_3	Class_4	Class_5	Class_6	Class_7	Class_8	Class_9
Class_9	96	5	3	3	3	13	7	31	830
Class_8	104	7	14	6	6	36	49	1423	47
Class_7	34	9	32	12	3	19	439	18	1
Class_6	94	11	9	39	4	2483	84	56	47
Class_5	1	4	2	2	535	1	2	0	0
Class_4	1	45	84	376	2	10	18	0	2
Class_3	4	307	988	200	2	0	85	4	10
Class_2	13	2010	754	299	12	4	102	5	25
Class_1	274	6	3	3	2	9	24	23	41

Figure 14: Tuned Random Forest (SMOTE) - Confusion Matrix - Observation Distribution

The patterns of misclassification seem to have not changed much when compared to the confusion matrix shown earlier. Therefore, the recommendation of having someone specifically check and correct misclassifications of products belonging to class two, three and four applies here as well.

7 Summary and Outlook

The final model chosen in the preceding section should allow the Otto group to accurately classify a large part of new products entered into their data base. However, there are some points that could be improved or looked at, which could lead to further gains in accuracy. These include and are not limited to:

Cross validation: Throughout the report a 5-fold cross validation was applied. This was done primarily to reduce computing times across the board. Applying a higher number of folds in cross validation may be able to slightly improve models, or the respective differences between train and test accuracy.

Tuning: one might do well by retuning the random forest trained on SMOTE data more extensively to find the optimal .mtry parameters. Additionally, removing the cap of 150 tree estimators per model may be able to slightly improve accuracy even further.

SMOTE: While applying SMOTE to the training data led to great results, trying different combinations of the prec.over and perc.under parameters, which indicate what percentage of observations are over-or undersampled may lead to even better results.

Sampling: The approach undertaken throughout the report was to apply sampling on the training data and then training the classifiers on that data. While this still leads to good results, it risks making cross validation less effective. Since the test fold will also consist of sampled data, results may not be accurate when compared to unprocessed data. To avoid this problem, one can include sampling method as an argument in *caret's* train function through including "sampling = "down"" for example. This generates results more consistent with test data. However, doing so would have required a complete rework and rerun of the code.

Keeping all this in mind, the model generated should act as a good starting point for the Otto group, and with some human assistance, should also do well in accurately classifying new products based on their 93 features. Once these new products are correctly classified, they can be added as new observations to the initial dataset, thereby giving the classifier more data to train on leading to further gains in accuracy.