Jacobs University

Semester project

Otto Group Product Clasisfication Challenge

Rahul Bhat
M.Sc. Data Engineering

supervised by
Dr. Prof. Adalbert F.X. Wilhelm

March 7, 2017

Contents

List of Figures

List of Tables

Abstract

Product classification is the process of classifying service products. In this process, the products are divided according to specific characteristics or in other words, it is the process of grouping the products into different categories based on their properties and relationships to other products. There is no such formal classification system exists and even for the identification of the products every industry has their own standardized methods of product classification [1]. Product classification can be done in a variety of perspectives and it depends on the consumer and the business products. Product classification contains a huge number of categories and millions of products and each product must be mapped to the corresponding category. Hence this requires a tremendous effort by integrating different predictive analytical techniques including data mining, statistical modeling and machine learning which helps the analysts to build a predictive model which would be able to distinguish between the product categories [10].

Before applying any analytical techniques, the first thing is to do data-preprocessing and preparation, which helps the analysts to make a good choice of predictive analytical technique to apply. So after exploring the data, we decided to use four different predictive analytical techniques, that are Random Forest, Neural Networks, Xgboost and Generalized Boosted Regression Models. The four techniques which are used in this project has different methods of usage, these different methods were used on the provided data set first and the best method in each technique was noted and used for comparison with other techniques for the same data set.

Keywords: Predictive Analytical Techniques, Product Classification

A Introduction

A.1 Why We Chose this Project

Data mining is an emerging discipline, but still, there are many issues and challenges in terms of technologies and methodologies. Data mining techniques are used in each and every field, such as data-mining techniques are adopted by many banks, retailers, industries and also used by many credit firms and because of this data mining has grown into a billion dollar business.

This project will help us to contribute to achieving what we have learned in our discipline and the techniques which we are using in this project will help us to find out how to retain knowledge and how to deal with the real data from the perspective of machine learning and then that knowledge can be used to help in future learning and problem solving.

A.2 Statement of the Problem

The industry and research product for this semester have been adopted from Kaggle. Kaggle was founded in 2010, as a platform for data science competitions which helps the participants to solve difficult problems and amplify the power of data science talent. The competition host provides the raw data and a description of the problem and data miners or statisticians from all over the world compete to produce the best models.

The research project which we have opted from kaggle is "Otto Group Product Classification challenge". This challenge was posted by the Otto Group on Mar 2015. The Otto Group is world's one of the biggest e-commerce companies, with subsidiaries in more than 20 countries. The Otto Groups sells millions of products worldwide every day, with several thousand products being added to their product line. Missing or incorrect product classifications are one of the biggest issues and challenges for all the industries because of this wrongly classified products often results in delays at customs, penalties for incorrect filing, delays in shipment clearance and loss of preferential duty rates. Due to the diverse global infrastructure, the identical products sometimes get wrongly classified and therefore the quality of the product analysis depends heavily on the ability to accurately cluster similar products. This is the only reason the Otto Group posted the challenge on kaggle.

A.3 Data Introduction and Objective

The dataset which is provided by the Otto Group "https://www.kaggle.com/c/otto-group-product-classification-challenge/data" has 93 features for more than 200,000 products. The purpose of this project is to build a predictive model to correctly classify products between 9 product categories (fashion, electronics etc.

The data set is already split non-randomly into two data sets, train data, and test data. The training data (training dataset) is used for supervised learning, in which the input data is given with the correct or expected output for every data row and the test data (testing dataset) is on which the model is applied.

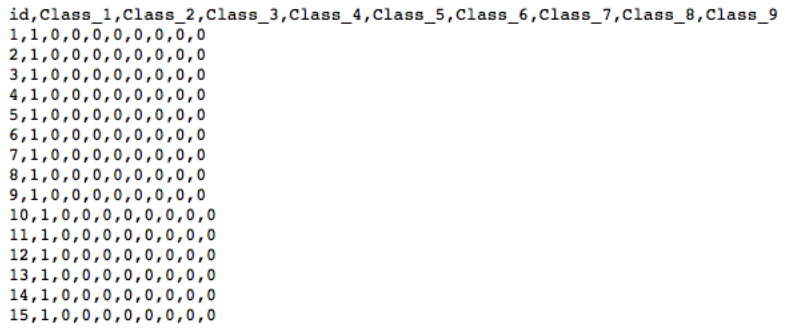
File descriptions

- train.csv - the training set
- test.csv - the test set

Data fields

- id - an anonymous id unique to a product
- feat_1, feat_2, ..., feat_93 - the various features of a product
- target - the class of a product

The final result (submission) should be in a particular format in which there would be a product id and all the classes with their associated probabilities. The sample submission files are shown below.



```
id,Class_1,Class_2,Class_3,Class_4,Class_5,Class_6,Class_7,Class_8,Class_9
1,1,0,0,0,0,0,0,0,0
2,1,0,0,0,0,0,0,0,0
3,1,0,0,0,0,0,0,0,0
4,1,0,0,0,0,0,0,0,0
5,1,0,0,0,0,0,0,0,0
6,1,0,0,0,0,0,0,0,0
7,1,0,0,0,0,0,0,0,0
8,1,0,0,0,0,0,0,0,0
9,1,0,0,0,0,0,0,0,0
10,1,0,0,0,0,0,0,0,0
11,1,0,0,0,0,0,0,0,0
12,1,0,0,0,0,0,0,0,0
13,1,0,0,0,0,0,0,0,0
14,1,0,0,0,0,0,0,0,0
15,1,0,0,0,0,0,0,0,0
```

Format of the File

The format of the file is "CSV" stands for "Common Separated Values" which is often used to exchange and convert data between various spreadsheet programs. There are cells inside such data file, which is separated by a special character, which usually is a comma and other characters can also be used as well. The first row of this data file contains the column names instead of actual data.

A.4 Area or Scope of Investigation

To gain a superficial knowledge and to see the nature of the datasets, preliminary analyses is done on the data sets before applying any technique. Preliminary analyses are very important in real-life analyses because it helps the analyst to gain knowledge about the data and making a choice of the technique to be used in the regression work. After analyzing the dataset, we have decided for the purpose of this work, we will use "R" for all of the analyses. R has its own libraries and with the help of these libraries, we will try to use more data mining techniques to get the better and improved result.

B Methodology: Approaches and Techniques

B.1 Data Visualization

The next step after data preparation is data visualization, that is the fastest and most useful way to learn and understand more about the data, which can yield better results. So more and more plots and graphs can be constructed to understand the data in depth. There are many ways in which one can understand data but one way to see and understand patterns from data is by means of visualization.

B.1.1 Visualization of Data using t-SNE

We have used "t-Distributed Stochastic Neighbor Embedding (t-SNE)" which is a famous and prize-winning technique and is most commonly used for the dimensionality reduction. If the dataset is a large real-world data set or high-dimensional dataset, for example, up to 20 million then "t-sne" is a good choice because that allows optimization, and produces significantly better visualizations. t-SNE is also good at creating maps and it gives each data point a location by revealing the structure and embedding relationships at many different scales [7].

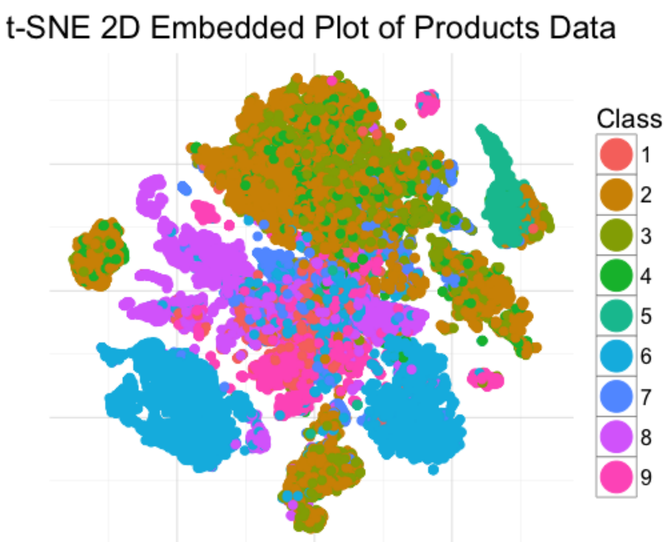


Figure 1: Training Data Sample (30,000). The colors of the points indicate the classes of the corresponding objects. The titles of the figures indicate the computation time that was used to construct the corresponding embeddings.

B.1.2 Principal Component Analysis (PCA) and K-means Clustering

Principal Component Analysis

We have used Principal Component Analysis (PCA) because it brings out strong patterns in a dataset and it helps to make data easy to explore and visualize. PCA finds a new coordinate system in which every point has a new (x,y) value.

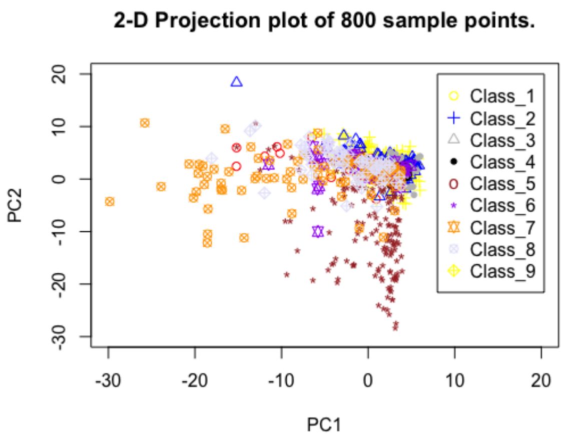


Figure 2: 2-D Projection Plot of 2000 Samples

K-means Clustering

K-means is a partitioning method which divides the data into k non-overlapping clusters, by using this method the objects of different clusters would be separated from the objects of the same cluster. The objects of the same cluster would be closer to each other and the objects of the different cluster would be dissimilar. As we can see from the Figure 2(b), a graphical representation in which there are objects with their interrelations and at the same time shows the clusters. This graphical representation allows us to imagine or picture the shape and size of the clusters and their relative positions.

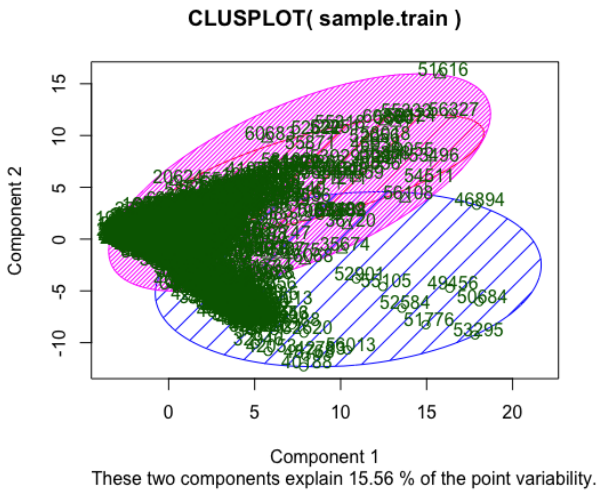


Figure 3: K-Means Clustering with 20 Clusters Samples

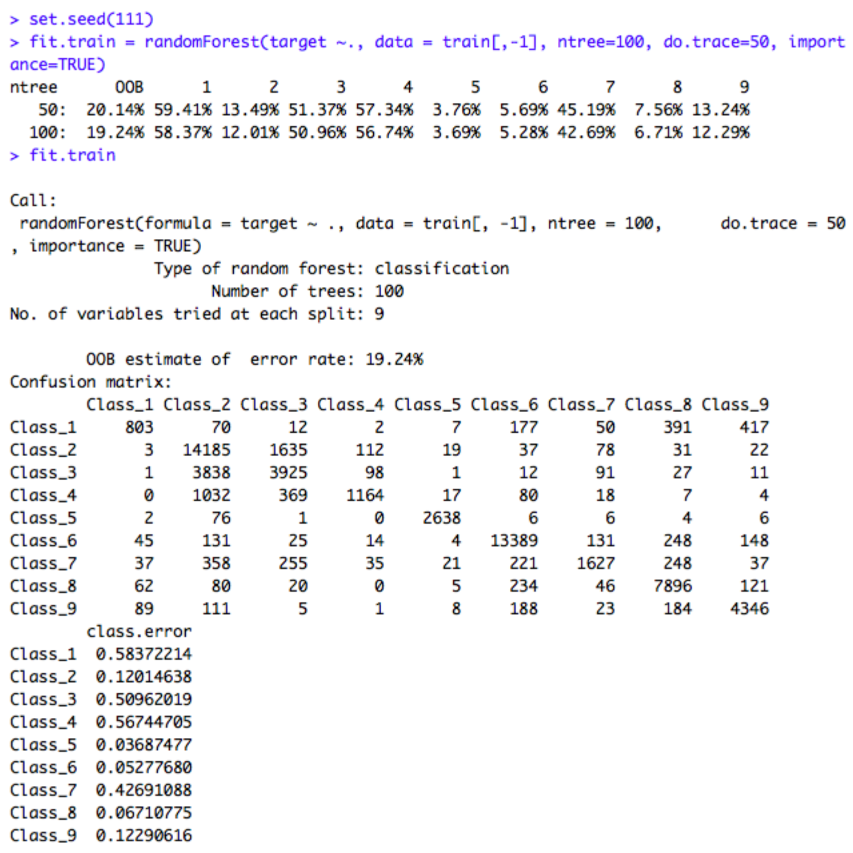
B.2 Random Forest

Random forest is a very popular method for predictive analytics. Random forest is an ensemble of decision trees. It is a type of “ensemble learning” technique for classification, regression and for some other tasks as well which extends on decision trees. Ensemble learning is a procedure which gives a prediction value, in this approach a team of predictive models is constructed to solve the given prediction task [5].
Random Forest grows many classification trees using a subset of the available input variables and their values. Each tree in the forest gives a classification by putting an input vector down step by step to each of the trees in the forest and at every step, a new object is formed from the input vector. It is like tree votes for that class and the forest chooses the one with the maximum number of votes [5].

B.2.1 Methodology

In this section, we have applied random forest approach. Random forests are like a body of trees that are built using a variant which is specially called "bagging". The important thing in any technique is to construct a model and for that random forests only uses a proportion of the predictors.

Before implementing random forest technique, we need a package called "randomForest" which contains a randomForest () function, along with that bagging can also be performed using this function because bagging is a special case of random forest. Random forest is performed as follows:



```
> set.seed(111)
> fit.train = randomForest(target ~., data = train[,-1], ntree=100, do.trace=50, import
ance=TRUE)
ntree      OOB      1      2      3      4      5      6      7      8      9
   50:  20.14% 59.41% 13.49% 51.37% 57.34%  3.76%  5.69% 45.19%  7.56% 13.24%
  100:  19.24% 58.37% 12.01% 50.96% 56.74%  3.69%  5.28% 42.69%  6.71% 12.29%
> fit.train

Call:
 randomForest(formula = target ~ ., data = train[, -1], ntree = 100,      do.trace = 50
, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 100
No. of variables tried at each split: 9

        OOB estimate of  error rate: 19.24%
Confusion matrix:
        Class_1 Class_2 Class_3 Class_4 Class_5 Class_6 Class_7 Class_8 Class_9
Class_1     803      70      12       2       7     177      50     391     417
Class_2       3   14185    1635     112      19      37      78      31      22
Class_3       1    3838    3925      98       1      12      91      27      11
Class_4       0    1032     369    1164      17      80      18       7       4
Class_5       2      76       1       0    2638       6       6       4       6
Class_6      45     131      25      14       4   13389     131     248     148
Class_7      37     358     255      35      21     221    1627     248      37
Class_8      62      80      20       0       5     234      46    7896     121
Class_9      89     111       5       1       8     188      23     184    4346
        class.error
Class_1  0.58372214
Class_2  0.12014638
Class_3  0.50962019
Class_4  0.56744705
Class_5  0.03687477
Class_6  0.05277680
Class_7  0.42691088
Class_8  0.06710775
Class_9  0.12290616
```

Figure 4: OOB estimate error and confusion matrix

First, a formula is used to fit the data for randomForest () function. After using the formula, we got the summary which is shown in the above figure (Figure 4). The summary contains a briefing of the variables and the trained model. It shows the type of random forest "classification", number of trees "100" and the number of the variables tried at each split "9". The OOB (out-of-bag) error is nothing but the complementary to the accuracy. The confusion matrix gives us the summary of how many cases were guessed right from our model. The confusion matrix is based on actual response variable and predicted value. In our case, the OOB error is 19.24%, which is equivalent to saying that the accuracy of our model is 80.76%. We can also manually select the number of trees by (ntree=100) and as the size of the tree increases the error rate decreases and vice versa.

Plot(fit.train)

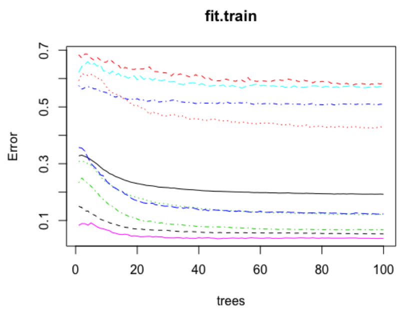


Figure 5: Error Rate vs Number of Trees

In Figure 5, we can see that, though the initial errors were higher, as the number of trees increased, the errors slowly dropped somewhat over all. The black line is the OOB "out of bag" error rate for each tree and the rest of the nine lines are our classes. When there are few trees the error rate is high, but as more trees are added the error rate decreases and eventually flatten out.

Random forests have many advantages but the one major advantage is that they can provide a measure of relative importance by ranking predictors based on how much they influence the response. Importance can be obtained by using the importance () function and we can also plot the importance using the varImpPlot () function.

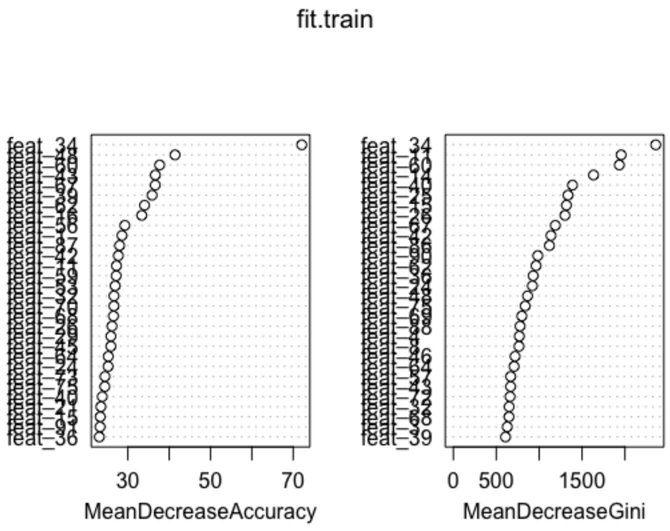


Figure 6: Variable Importance Plot

The variable importance plot shows above (Figure 6) is an output of the random forest algorithm. In our dataset, there are 93 features and every feature is shown in the above plot and the importance of each feature. On the Y-axis, it shows each feature and on the X-axis, it shows how important that feature is (Importance).The variables are ordered from top-to-bottom according to their importance, which is given by the position of the dot on the X-axis. The main advantage of this plot is that it shows us which variable we could use for the data analysis and this tool also helps us in reducing the number of variables for other data analysis technique [6].

The results have been saved and after learning, our machine can make predictions from these results. We will use a generic predict () function to our testing dataset. R will then take the new data frame, process the variables according to the random forest formula and give out a result for each row of the new data frame.

B.2.2 Packages

- library(caret)
- library(ggplot2)
- library(randomForest)

These are the list of packages which are used for "Random Forest".

B.2.3 Observations and Parameters Change

There are three tuning parameters which are important when building random forests, that are node size, the number of trees, and the number of predictors sampled at each split. If we tune these parameters carefully, it can prevent extended computations with little gain in error reduction. For example, if we reduce the size of the tree manually, the error rate will increase and vice versa, shown below in the table 1.

Tree Size	Error Rate
50	20.17%
100	19.24%
150	18.94%
200	17.18%

Table 1: OOB Estimate Error vs Tree Size

B.3 Xgboost

Xgboost is short for eXtreme Gradient Boosting package. Xgboost is a library which is designed and optimized for boosting tree algorithms. Extreme Gradient Boosting (xgboost) is same as gradient boosting framework but Xgboost is more efficient, flexible and portable. It is the package which is used to solve data science problems which include both linear model solver and tree learning algorithms.

B.3.1 Parameters

There are various parameters used in Xgboost model : general parameters, booster parameters, and task parameters.

- General parameters are used to guide the overall functioning, it refers to which booster is used for boosting and the commonly used are the tree or linear model.
- Booster parameters depends on the booster which has been chosen.
- Task parameters are used to guide the optimization performed and task parameters that decide on the learning scenario, for example, different tasks may use different parameters.

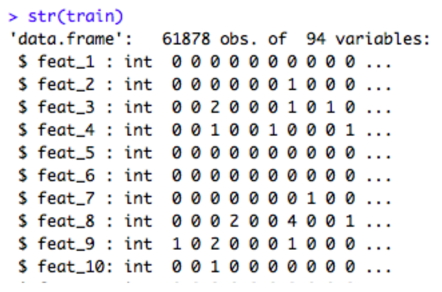
These parameters are used to improve the model and for that parameter tuning is must.

B.3.2 Methodology

The main idea behind Xgboost is, it divides the training data into parts and then Xgboost will retain the first part and use it as the test data. After this, it will reintegrate the first part to the training dataset and retain the second part and it goes on.

The first thing we have to do before building the model and tuning is data pre-processing because Xgboost only works with numeric vectors and doesn't accept data frames. So, first, we will inspect our data by using the following command.

```
str(train)
```



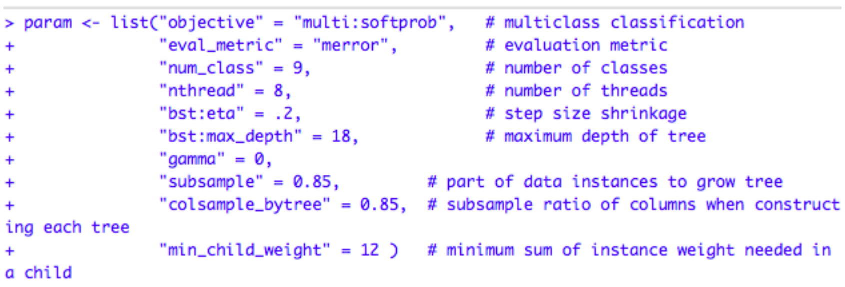
```
> str(train)
'data.frame':   61878 obs. of  94 variables:
 $ feat_1 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ feat_2 : int  0 0 0 0 0 0 1 0 0 0 ...
 $ feat_3 : int  0 0 2 0 0 0 1 0 1 0 ...
 $ feat_4 : int  0 0 1 0 0 1 0 0 0 1 ...
 $ feat_5 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ feat_6 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ feat_7 : int  0 0 0 0 0 0 0 1 0 0 ...
 $ feat_8 : int  0 0 0 2 0 0 4 0 0 1 ...
 $ feat_9 : int  1 0 2 0 0 0 1 0 0 0 ...
 $ feat_10: int  0 0 1 0 0 0 0 0 0 0 ...
```

Xgboost only allows dense and sparse matrix as an input and as we can see our data, it's a data frame. Therefore, the first thing we need to do is to convert that data frame into a matrix form and all other forms of data into numeric vectors as shown below.

```
target = as.matrix(as.numeric(train$target)-1)
train.xg = as.matrix(train.xg)
mode(train.xg) = 'numeric'
```

In the above code first we converted our train dataset to numeric mode and then we have bound our target column and train data as the columns of a matrix, then we removed the "ID" column because each column in a dataset represents a feature measured by an integer and we know that the first column (ID) doesn't contain any useful information. To let the algorithm focus on real stuff, we deleted it and make that "NULL". Factors and ordered factors are replaced by their internal codes. We did the same thing to our test data as well.

Afer data pre-processing the next part is to choose the parameters. How we have taken the parameters and their values are shown below.



```
> param <- list("objective" = "multi:softprob",    # multiclass classification
+               "eval_metric" = "merror",          # evaluation metric
+               "num_class" = 9,                   # number of classes
+               "nthread" = 8,                     # number of threads
+               "bst:eta" = .2,                    # step size shrinkage
+               "bst:max_depth" = 18,              # maximum depth of tree
+               "gamma" = 0,
+               "subsample" = 0.85,          # part of data instances to grow tree
+               "colsample_bytree" = 0.85,   # subsample ratio of columns when construct
ing each tree
+               "min_child_weight" = 12 )    # minimum sum of instance weight needed in
a child
```

Figure 7: Choosing Parameters

We have to set these parameters before running Xgboost. let us see what these parameters are and why they are important.

(a) **Learning Task Parameters**

(i) **objective = "multi:softprob" or "multi:softmax"**
This parameter is used for multiclass classification. We can also use "softmax". The difference between them is It's output which separates them. In softmax, we get the class with the maximum probability as output. In softprob, same as "softmax" but output a vector of ndata * nclass which can be further reshaped to ndata, nclass matrix. So, the result contains the predicted probability of each data point belonging to each class. We can also say that in "softprob" we get a matrix with a probability value of each class we are trying to predict. If we are using "softmax" parameter we also need to set num_class(number of classes) [4].

(ii) **eval_metric**
It is an evaluation metrics used for validation data. In this case, a default metric is assigned according to the objective. The objective could be rmse for regression, error for classification, mean average precision for ranking [4].

(b) **Parameters for Tree Booster**

(i) **max_depth**
It is the maximum depth of a tree and its default value is "6". Increasing the value of maximum depth makes the model more complex. The range is $[1, \infty]$ [4].

(ii) **eta**

The default value of eta is "0.3". It is a step size shrinkage used in an update to prevents overfitting. We can directly get the weights of new features after each boosting step and eta actually shrinks the feature weights to make the boosting process more conservative.The range is [0,1] [4].

(iii) **gamma**

The default value of gamma is 0. Gamma is a minimum loss reduction required to make a further partition on a leaf node of the tree. The larger the value of gamma, the more conservative the algorithm will be. The range is $[0, \infty]$ [4].

(iv) **subsample**

The default value of subsample is "1". The subsample is the ratio of the training instance. If we set the value of subsample to 0.5, it means that XGBoost randomly collected half of the data instances to grow trees and this will prevent overfitting. The range is [0,1] [4].

(v) **colsample_bytree**

The default value of colsample_bytree is "1". Colsample_bytree is a subsample ratio of columns when constructing each tree. The range is [0,1] [4].

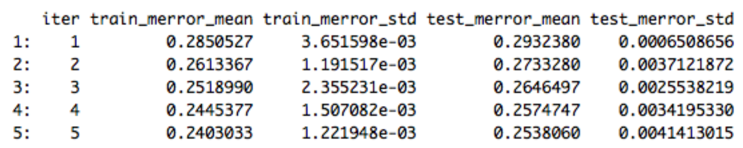
(vi) **min_child_weight**

It is a minimum sum of instance weight needed in a child. If the partitioning results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning but in linear regression, this simply corresponds to a minimum number of instances needed to be in each node. Larger the value of min_child_weight, the more conservative the algorithm will be. The range is $[0, \infty]$ [4].

The main idea behind Xgboost is, it divides the training data into parts and then Xgboost will retain the first part and use it as the test data. After this, it will reintegrate the first part to the training dataset and retain the second part and it goes on.

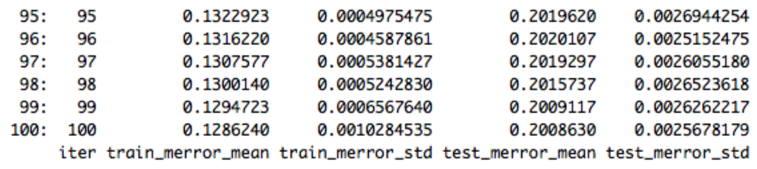
Now after choosing the parameters, the next step is cross-validation. In cross-validation, we have to set the seed value first and after that, we have to choose the number of decision trees in the final model (nrounds) and in how many parts we want to divide the train data into for the cross-validation (nfold). We can edit the values for nfold and nrounds to get better results. Now the next step is to run the xgb.cv () function which is used for cross-validation with the command shown below.

```
xgbst.cv = xgb.cv(param=param, data= train.xg, label=target, nfold=3, nrounds=100)
```



```
    iter train_merror_mean train_merror_std test_merror_mean test_merror_std
 1:    1         0.2850527     3.651598e-03        0.2932380    0.0006508656
 2:    2         0.2613367     1.191517e-03        0.2733280    0.0037121872
 3:    3         0.2518990     2.355231e-03        0.2646497    0.0025538219
 4:    4         0.2445377     1.507082e-03        0.2574747    0.0034195330
 5:    5         0.2403033     1.221948e-03        0.2538060    0.0041413015
```

We can see that, the train_merror is decreasing (For 100 nrounds). Each line shows how well the model explains our data (lower is better).

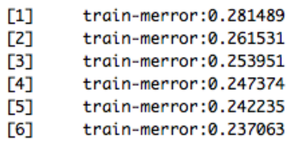


```
 95:   95         0.1322923     0.0004975475        0.2019620    0.0026944254
 96:   96         0.1316220     0.0004587861        0.2020107    0.0025152475
 97:   97         0.1307577     0.0005381427        0.2019297    0.0026055180
 98:   98         0.1300140     0.0005242830        0.2015737    0.0026523618
 99:   99         0.1294723     0.0006567640        0.2009117    0.0026262217
100:  100         0.1286240     0.0010284535        0.2008630    0.0025678179
    iter train_merror_mean train_merror_std test_merror_mean test_merror_std
```

The return value is a "data.table" containing the measurements on training and testing folds. We can easily track the best number of rounds. As we can see at 100^{th} iteration the error went down. As the number of rounds increases, the error rate decreases. The main point is to keep an eye on the "test.merror" which is more important than train.merror.

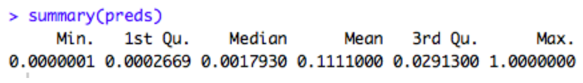
The function xgb.cv () returns a data.table object containing the cross-validation results. This is helpful for choosing the correct number of iterations. After this the next step is to train our model using xgboost () function. The command is same instead of xgb.cv () function we will use xgboost () function.

```
bst <- xgboost(param=param, data= train.xg, label=target, nrounds=100)
```



```
[1]     train-merror:0.281489
[2]     train-merror:0.261531
[3]     train-merror:0.253951
[4]     train-merror:0.247374
[5]     train-merror:0.242235
[6]     train-merror:0.237063
```

As we can see the train-merror is decreasing when we trained our model. After training the model, the next step is to predict the final result using the predict () function in which we will use our test data. The result is shown below.



```
> summary(preds)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
0.0000001 0.0002669 0.0017930 0.1111000 0.0291300 1.0000000
```

Xgboost offers a better representation and that is feature importance. Feature importance is about averaging the gain of each feature for all split and all trees. For feature importance the function "xgb.plot.importance" is used. In the feature importance below (Figure 8), we can see the first 10 most important features. K-means clustering is applied to each group feature by importance. The best part of feature Importance plot is that we get to know that which feature is less important and for better results, we can remove that less important feature. Below is the code to plot the feature importance.

```
importance_matrix <- xgb.importance(names, model = bst)
```

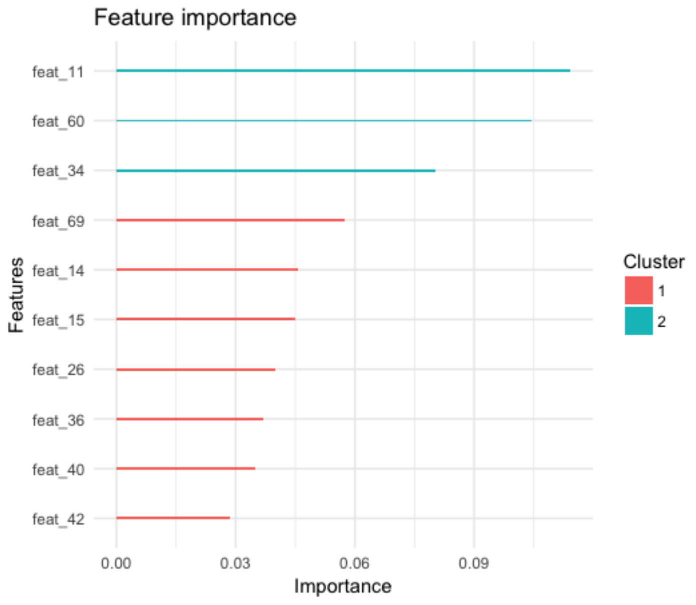


Figure 8: Feature Importance

B.3.3 Packages

- library(caret)
- library(corrplot)
- library(xgboost)
- library(knitr)
- library(ggplot2)

Above are the list of packages which we have used for "xgboost"

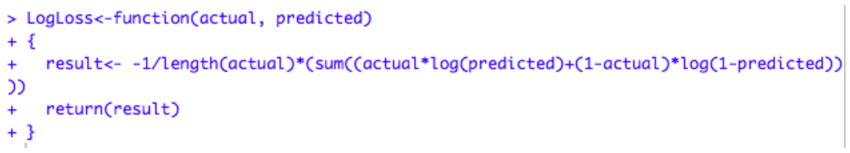
B.3.4 Observations and Parameter Change

We have also calculated the Logarithmic Loss (Log Loss), it is a classification loss function that is used for an evaluation. Log Loss quantifies the accuracy of a classifier by penalizing false classifications [8]. The classifier calculates the Log Loss by assigning a probability to each class rather than yielding the most likely class.

Mathematically Log Loss is defined as:

logloss= $-\frac{1}{N}\Sigma\sum_{i=1}^{n}\sum_{j=1}^{j} y_{ij}\ log(P_{ij})$

where N is the number of examples, M is the number of classes, and y_{ij} is the predicted probability that observation "i" belongs to class "j". To calculate Log Loss, we have used the log loss function, shown below.



```
> LogLoss<-function(actual, predicted)
+ {
+   result<- -1/length(actual)*(sum((actual*log(predicted)+(1-actual)*log(1-predicted))
))
+   return(result)
+ }
```

To get the Log Loss result, we have to use the following command.

print(LogLoss(actual, preds))

We changed the parameters and recorded the values and we observed that there was an improvement when we decreased the value of shrinkage and increased the tree size. The results are shown in the table below.

Xgboost Parameters	Validation (log loss)
num_class = 9, shrinkage = 0.2, tree size = 200, gamma = 1.5, subsample = 0.85, colsample_bytree = 0.85	0.47762
num_class = 9, shrinkage = 0.1, tree size = 400, gamma = 1, subsample = 0.85, colsample_bytree" = 0.85	0.46986
num_class = 9, shrinkage = 0.05, tree size = 500, gamma = 0.5, subsample = 0.85, colsample_bytree = 0.85	0.45954

Table 2: Paramter Change and Recorded Result

B.4 Generalized Boosted Regression Models (gbm)

Gradient Boosting is one of the boosting algorithms and this technique works on the principal of ensemble. Gradient Boosting computes each level of the tree and when computing is done, it combines the trees and develops an ensemble tree-based models. Gradient boosting is very efficient and powerful boosting technique, it gives an improved prediction accuracy by combining the sets of weak learners. The main difference between AdaBoost and Gradient Boosting is that in AdaBoost, the weights are added at each stage and in Gradient Boosting technique, Gradients are used to identify the weakness [2]. There are many advantages of using Gradient Boosting but the most important feature of Gradient Boosting is, it can explicitly handles NA's and more factor levels than Random Forest.

B.4.1 Methodlogy

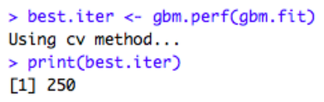
Boosting is nothing but adding the number of trees which are structured to improve fit. Gradient boosting always comes with a cross-validation (cv) option, which is used to determine the optimum number of trees. If there are variables in the data.frame which is not used in the model then an error will appear. So we have first selected the data in the code and then only we can run our model.

The model parameters which we have chosen are the number of trees, shrinkage, the number of folds and depth. We can choose other parameters also to improve our final result. After choosing the parameters we can fit our model using gbm () function and use the cross-validation procedure to find the best number of trees for prediction. We can set the seeds randomly to make sure that the gbm function ensures reproducibility for the distributed cross-validation folds. When we are modeling the data, there is always a big concern and that is overfitting, so we have used a 10-fold cross validation and the number of trees we are using to fit our model are 250. The number of trees should be high enough to get the better result. The code is shown below.

```
gbm.fit <- gbm(target  ., data=train, distribution="multinomial", n.trees=250, shrinkage=0.02,
interaction.depth=8, cv.folds=10))
```

The above code shows that how we have fitted our model and how we have chosen the parameters and their values. We can change the values of parameters anytime to get better results.

We have used the gbm.perf() function, this function is used to estimate "the optimal number of boosting iterations for a gbm object" and we have also used gbm.Summary() to summarize the relative influence of each variable in the gbm object. Below is a graphical summary of the relative influence of each variable.



```
> best.iter <- gbm.perf(gbm.fit)
Using cv method...
> print(best.iter)
[1] 250
```

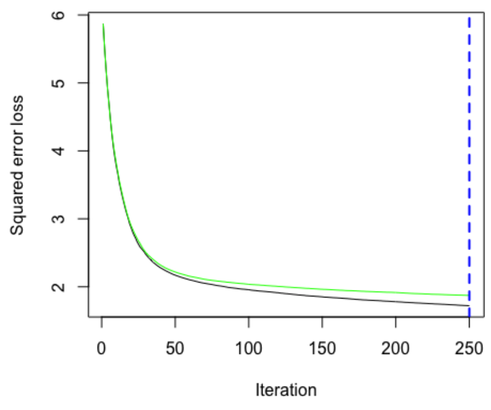


Figure 9: Optimal Number of Boosting Iterations

In Figure 9, we can see two curves black and green. The black curve shows our model deviance on training data as a function of the number of iterations while the green curve shows model deviance on a holdout. A point is selected by the algorithm and from that point, the holdout deviance begins to increase again. In our case as we have seen above also the optimal number of boosting iterations is for 250 trees and the depth we have selected is 8 and that is essentially the number of variables in our model.

```
summary(gbm.fit,
cBars=length(gbm.fit$target),
n.trees=gbm.fit$n.trees,
plotit=TRUE,
order=TRUE,
method=relative.influence,
normalize=TRUE)
```

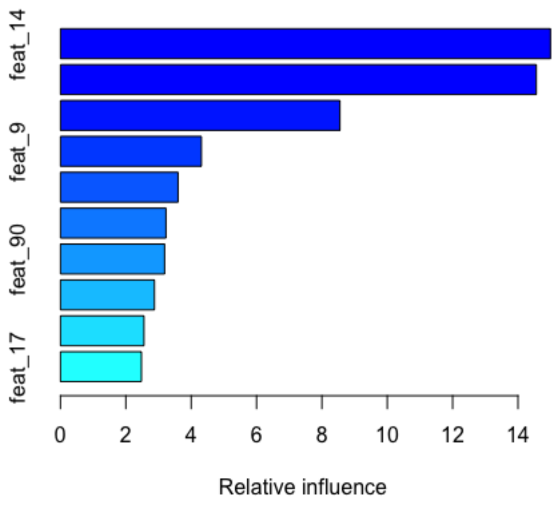


Figure 10: Feature Importance

In the above figure (Figure 10) we can see the relative influence, it shows the contribution of each variable in minimizing the loss function. It is calculated by taking the average number of times a variable is selected for splitting weighted by the squared improvement to the model as the result of each split. After this, it is then scaled, so that the values will sum to 100. In our case feature 14 (feat_14) having the most explanatory power.

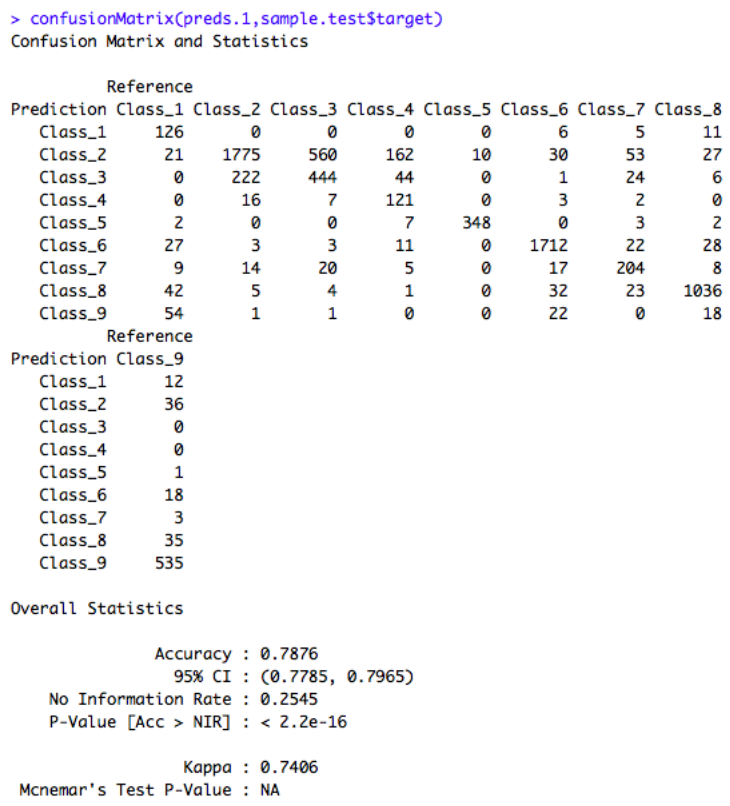
We can tune the parameters and look for much better results. The next step after fitting the model is to test the boosting model on the sample test data. We can use the best iteration in the predict function () which we got earlier that is 250. We can replace the number of trees by best iteration, shown in the below code.

gbm.predict <- predict(gbm.fit, test, n.trees=best.iter, type="response")

In the above code, we have used the predict () function on our testing data and the result of our fitting model (i.e gbm.predict). The result of best iteration is used for the number of trees.

We have taken a testing sample of 8000 and created a confusion matrix of predictions vs actuals. The confusion matrix gives the result in a tabular summary of the actual class labels vs. the predicted ones. Let us take a look at the confusion matrix figure (Figure 12) shown below, the first table shows the contents of our matrix. Each column holds the reference or the actual data and within each row is the prediction. The diagonal of the confusion matrix represents instances where our observation correctly predicted the class of the item.

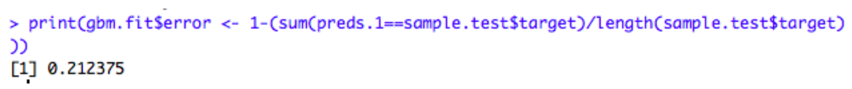
There is overall statistics just below the confusion matrix which contains summary statistics for the results. Overall accuracy is calculated at just over 78% with a p-value of 2*10^{-}16, or 0.00000000000000022. As we can say that our classifier has done a good job of classifying items.



```
> confusionMatrix(preds.1,sample.test$target)
Confusion Matrix and Statistics

          Reference
Prediction Class_1 Class_2 Class_3 Class_4 Class_5 Class_6 Class_7 Class_8
   Class_1     126       0       0       0       0       6       5      11
   Class_2      21    1775     560     162      10      30      53      27
   Class_3       0     222     444      44       0       1      24       6
   Class_4       0      16       7     121       0       3       2       0
   Class_5       2       0       0       7     348       0       3       2
   Class_6      27       3       3      11       0    1712      22      28
   Class_7       9      14      20       5       0      17     204       8
   Class_8      42       5       4       1       0      32      23    1036
   Class_9      54       1       1       0       0      22       0      18
          Reference
Prediction Class_9
   Class_1      12
   Class_2      36
   Class_3       0
   Class_4       0
   Class_5       1
   Class_6      18
   Class_7       3
   Class_8      35
   Class_9     535

Overall Statistics

               Accuracy : 0.7876
                 95% CI : (0.7785, 0.7965)
    No Information Rate : 0.2545
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.7406
 Mcnemar's Test P-Value : NA
```

Figure 11: Confusion Matrix and Overall Statistics

The last thing to do is to determine the error rate for our model using the formula shown below.



```
> print(gbm.fit$error <- 1-(sum(preds.1==sample.test$target)/length(sample.test$target)
))
[1] 0.212375
```

B.4.2 Observations

We created a sample training dataset (8000 samples) and testing data-set (3000 samples) and then we changed the parameters and observed the changes in the accuracy and error rate. Observations are shown below in the table.

Gradient Boosting Parameters	Accuracy
distribution="multinomial", n.trees=750, shrinkage=0.1, interaction.depth=10, cv.folds=2	79.89
distribution="multinomial", n.trees=500, shrinkage=0.01, interaction.depth=8, cv.folds=3	79.10
distribution="multinomial", n.trees=250, shrinkage=0.02, interaction.depth=8, cv.folds=2	78.76

Table 3: Paramter Change and Recorded Results

B.5 Neural Networks

- Neural Networks is machine learning technique.
- Neural networks are non-linear statistical data modeling tools.
- Neural networks are the artificial systems which are sophisticated, perhaps intelligent.
- Neural networks perform the computations same as the human brain routinely performs, and thereby possibly enhance understanding of the human brain.
- Neural networks are very good in recognizing patterns and good at fitting non-linear functions.

Neural networks need training sessions in which it adapts itself based on examples. After completion of the training sessions, the neural networks is able to relate the problem data to the solutions and then it offers a viable solution to a brand new problem. Neural networks can also generalize and handle incomplete data. Their ability to learn by example makes them very flexible and powerful. There is no need to devise an algorithm in order to perform a specific task [9].

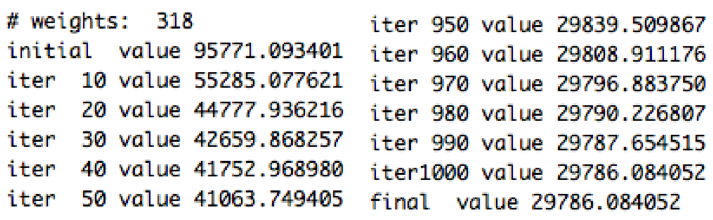
Neural networks are tools that have application in many areas and that is why neural networks are used in the aerospace, automotive, banking, defense, electronics, entertainment, financial, insurance, manufacturing, oil and gas, robotics, telecommunications, and transportation industries [3].

B.5.1 Methodology

To start with neural networks, we need two packages that are: nnet and neuralnet. First, we have to remove ID column from our training dataset before fitting our model by using neuralnet () function.

fi.nnet <- nnet(target ~, data=train2, size = 3, decay = 5e-4, maxit = 1000, linear.output FALSE, threshold=0.01)

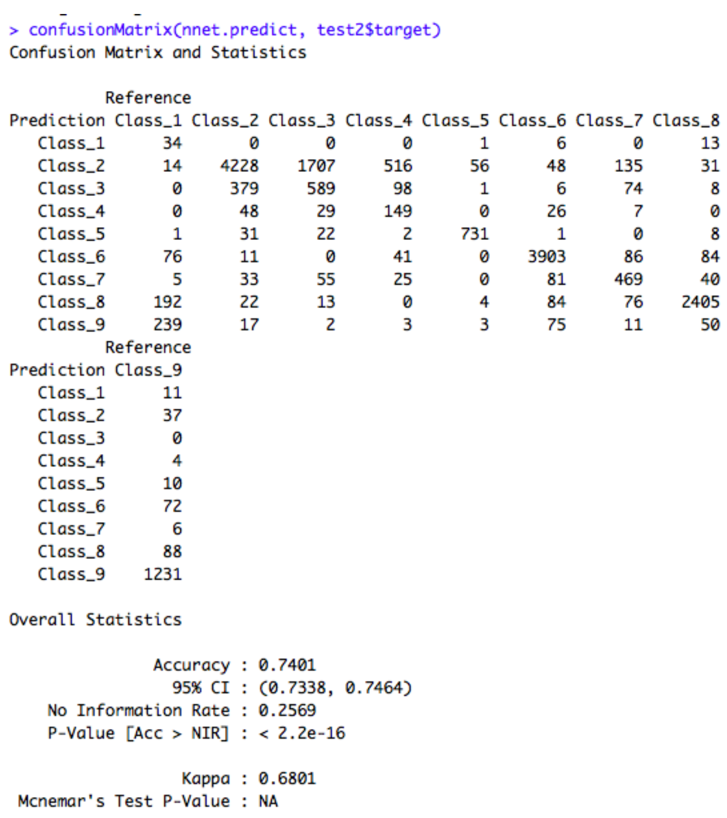
In the above code, we can see the parameters which we have used. We can add or remove the parameters and the check for the better outcomes. We have used maximum iteration value to 1000, the default value of iteration is 100. As the iteration increases, the outcome would be better. Decay is a parameter for weight decay. The default value for decay is 0. The sample of the function nnet () for 1000 iterations are shown below.



```
# weights:  318                      iter 950 value 29839.509867
initial  value 95771.093401          iter 960 value 29808.911176
iter  10 value 55285.077621          iter 970 value 29796.883750
iter  20 value 44777.936216          iter 980 value 29790.226807
iter  30 value 42659.868257          iter 990 value 29787.654515
iter  40 value 41752.968980          iter1000 value 29786.084052
iter  50 value 41063.749405          final  value 29786.084052
```

We can see that the initial value was 95771.09 and after the iteration stops at 10000, the final value went down to 29786.08.

We also created a confusion matrix which contains the summary statistics for the final result. The confusion matrix is shown below in the figure (Figure 14). The overall accuracy of the neural network (1000 iterations) is 74.01%.



```
> confusionMatrix(nnet.predict, test2$target)
Confusion Matrix and Statistics

          Reference
Prediction Class_1 Class_2 Class_3 Class_4 Class_5 Class_6 Class_7 Class_8
   Class_1      34       0       0       0       1       6       0      13
   Class_2      14    4228    1707     516      56      48     135      31
   Class_3       0     379     589      98       1       6      74       8
   Class_4       0      48      29     149       0      26       7       0
   Class_5       1      31      22       2     731       1       0       8
   Class_6      76      11       0      41       0    3903      86      84
   Class_7       5      33      55      25       0      81     469      40
   Class_8     192      22      13       0       4      84      76    2405
   Class_9     239      17       2       3       3      75      11      50
          Reference
Prediction Class_9
   Class_1      11
   Class_2      37
   Class_3       0
   Class_4       4
   Class_5      10
   Class_6      72
   Class_7       6
   Class_8      88
   Class_9    1231

Overall Statistics

               Accuracy : 0.7401
                 95% CI : (0.7338, 0.7464)
    No Information Rate : 0.2569
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.6801
 Mcnemar's Test P-Value : NA
```

Figure 13: Confusion Matrix and Overall Statistics

B.5.2 Packages

- library(nnet)
- library(neuralnet)

The packages which are required for neural networks are "nnet" and "neuralnet". Package "nnet" is used to generate a class indicator function from a given factor. The "neuralnet" package is used to visualize the generated model and shows the found weights.

B.5.3 Observations and Parameters Change

Neural Networks Parameters	Accuracy
size = 8, rang = 0.3, decay = 5e-4, maxit = 1230, linear.output = FALSE,threshold=0.01	76.52%
size = 3, rang = 0.1, decay = 5e-4, maxit = 1000, linear.output = FALSE,threshold=0.01	74.01%
size = 5, rang = 0.2, decay = 5e-4, maxit = 700, linear.output = FALSE,threshold=0.01	70.56%
size = 4, rang = 0.3, decay = 5e-4, maxit = 500, linear.output = FALSE,threshold=0.01	68.12%

Table 4: Paramter Change and Recorded Results

We observed that the accuracy rate increased when we increased the number of iterations. The computation time of Neural Networks is better than the other techniques, but the accuracy rate is not as high than others.

C Final Result

Techniques	Accuracy
Random Forest 200 Trees	82.82%
Gradient Boosting 750 Trees	79.89%
Neural Networks 1230 Iterations	76.52%

Table 5: Accuarcy of Classification for each Algorithm

We have obtained the results from the three techniques and created a final conclusion table based on their accuracy. The three techniques are Random Forest, Gradient Boosting and Neural Networks. As we can see that the Random Forest algorithms obtained the highest accuracy of 82.82% (number of trees = 200). if we can increase the number of trees, we will get the highest accuracy. Gradient Boosting technique obtained the second highest accuracy, while Artificial Neural Networks obtained the third, significantly low as compared to the rest. The observations and analysis which we have conducted shows that the Random Forest is the better classification technique as compared to the Gradient Boosting and Artifical Neural Networks for classifying the data.

We have also obtained the result for Xgboost technique which is based on Log Loss method because Log Loss is often used as an evaluation metric in kaggle competitions and Xgboost has a parameter called "eval_metric' which provides the options to optimize our model with respect to Log Loss as well as multiclass Log Loss also known as (MLogLoss). So we tried the Log Loss function in Xgboost technique. The results are shown in the table 2.

D Conclusion and Discussion

We trained four models Random Forest, Xgboost, Gradient Boosting and Neural Networks. We also changed the various parameters and recorded the best results. Tree-based model is simple yet relatively accurate. The accuracy rate of Random Forest was highest among all but Random Forest was also slowest than the rest of the techniques.

Between Xgboost and Gradient Boosting (GBM), Xgboost performed faster and better than Gradient Boosting. Xgboost is highly efficient, flexible and portable. The most important feature of Xgboost is it can automatically do parallel computation on a single machine. If there is a large number of data then Xgboost will give the better performance but we have to keep on changing the parameters to check for the better result which is very time-consuming.

We also noticed that the neural networks were able to risk more with high probabilities and worked much better in the minority classes, While Gradient Boosting was more conservative in difficult instances. Data visualization is the fastest and most useful way to learn and understand more about the data, which can yield better results. So more plots and graphs could be constructed to understand the data in depth. Some other techniques can also be used to predict the final result.

We can also improve our model and yield better results by removing the least important features from our dataset. So, we can conclude that the accuracy rate can be improved with the change in parameters. As we increase the number of trees or iterations, we will get the better results but increasing the number of trees also slows down the computation time.

References

[1] Hitesh Bhasin. Marketing management articles. June 29, 2016.

[2] Michael Bowles. *Machine learning in Python: essential techniques for predictive analysis.* John Wiley & Sons, 2015.

[3] Martin T Hagan, Howard B Demuth, Mark H Beale, and Orlando De Jesús. *Neural network design*, volume 20. PWS publishing company Boston, 1996.

[4] AARSHAY JAIN. Complete guide to parameter tuning in xgboost. MARCH 1, 2016.

[5] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[6] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. Understanding variable importances in forests of randomized trees. In *Advances in neural information processing systems*, pages 431–439, 2013.

[7] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[8] R-Bloggers. 2015.

[9] Yashpal Singh and Alok Singh Chauhan. Neural networks in data mining. *Journal of Theoretical and Applied Information Technology*, 5(6):36–42, 2009.

[10] Steffen Staab and Rudi Studer. *Handbook on ontologies.* Springer Science & Business Media, 2013.