Individual Assignment

Statistics and Machine Learning

Wijayaweera, Dinu
Dinu.wijayaweera@ieseg.fr

Individual Assignment

Statistics and Machine Learning

Master of Big Data Analytics for Business

Dinu Wijayaweera

Contents

Task 1: Select 5 machine learning algorithms and explain the mechanisms

1. Logistic Regression

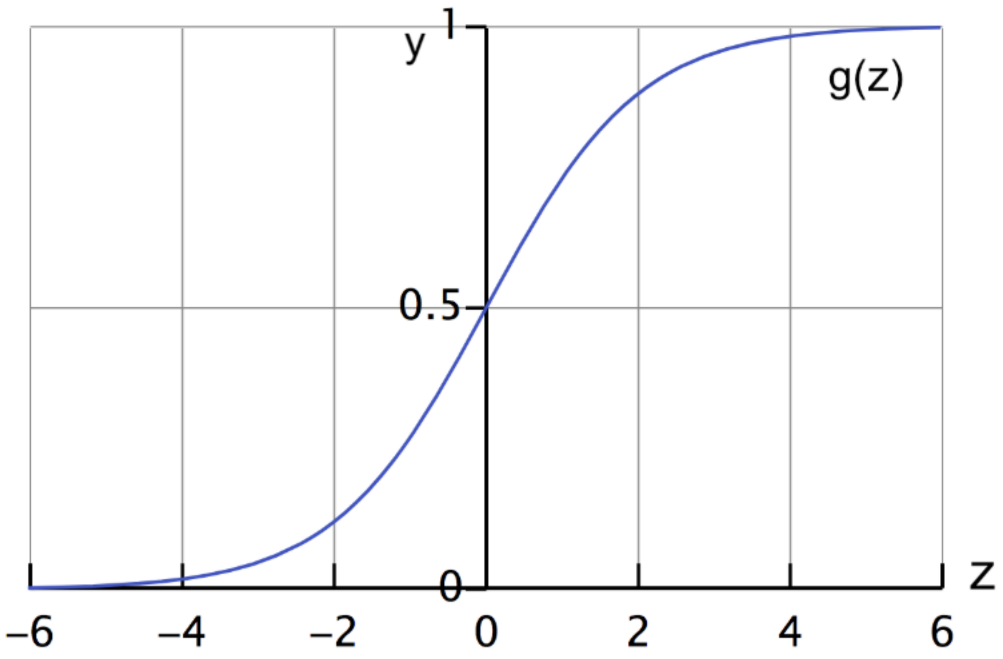
Introduction

Logistic Regression is a widely used model in machine learning and is used to predict the probability of a categorical dependent variable. Prior to explaining about logistic, the linear regression model would be explained to show how Logistic Regression came into being.

Linear Regression the dependent variable would be Continuous and measureable on ratio or scale. Linear Regression is a very powerful statistical technique and can be used to generate insights on consumer behaviour, understanding business and factors influencing profitability. However, Linear Regression would have trouble in situations where the response is only two values or ranges between

0,1. Logistic Regression would come into play to overcome such issues. In order to limit the responses between 0 and 1 the Sigmoid Function is applied for the linear function.

$$\textit{Sigmoid Function}: \quad g(z) = \frac{1}{1 + e^{(-z)}}$$



Logistic regression can be known as a transformation of the linear method, but the predictions are transformed using the logistic function. (Li, 2017)

The Logistic Regression Algorithm is used for Binary classification problems (Prabhakaran, 2017) such as Spam detection, Predicting if a user will buy a product or not.

There are different types of logistic regression models.

- Multinomial Logistic Regression

The Model: $$\ln\left(\frac{P_h}{P_j}\right) = \beta_{0h} + \beta_{1h}X_1 + \beta_{2h}X_2 + \ldots + \beta_{kh}X_k$$

Where j is the number of categories
h=1 to j-1
k is the number of predictors

- Binary Logistic Regression

The Model: $$Ln\left(\frac{P}{1-P}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_k X_k$$

$$Odds(Y=1) = e^{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_k X_k}$$

- Ordinal Logistic Regression/Proportional odds model

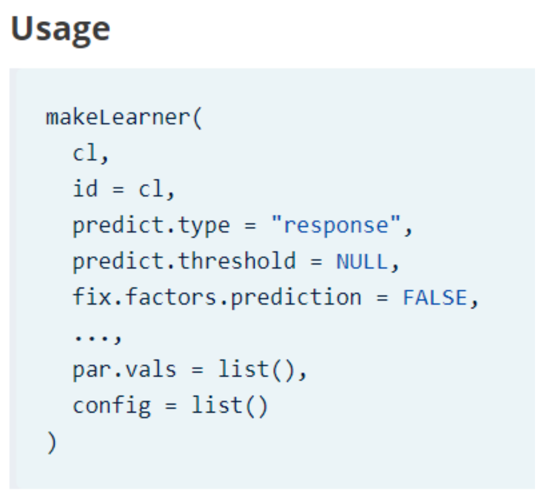
The Model: $$\ln\left(\frac{F_{ij}}{1-F_{ij}}\right) = \beta_{0j} + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_k X_k \qquad F_{ij} = \sum_{m=1}^{j} P_{im}$$

Maximum Likelihood Estimation (MLE) is used by both Multinomial and Binary Logistic Regression models as a learning algorithm. (Chatterjee, 2019) This is used to optimize the best values for the coefficients for training of the data.

Logistic Regression uses the stochastic Gradient descent to update the model until it converges. The same concept is used for Nueral Networks as well.

In order to build the logistic regression the users are required to set the family='binomial' in the glm function. 'glm' stands for 'Generalized Linear Models' and is capable of creating many regression models.

The model can be defined using the makeLearner function as well using the MLR package.

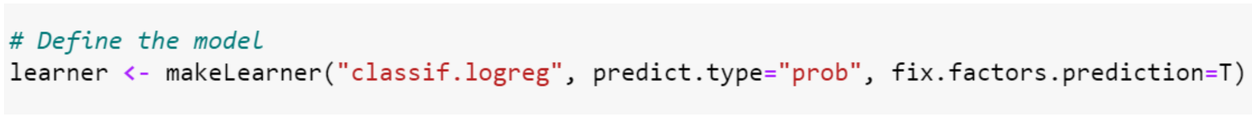


Usage

```
makeLearner(
  cl,
  id = cl,
  predict.type = "response",
  predict.threshold = NULL,
  fix.factors.prediction = FALSE,
  ...,
  par.vals = list(),
  config = list()
)
```

Figure 1: Source 'https://www.rdocumentation.org/packages/mlr/versions/2.17.0/topics/makeLearner'

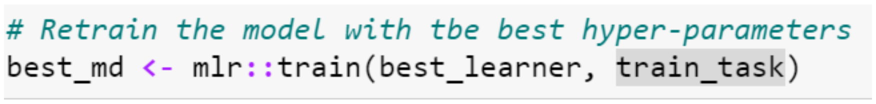
The below image shows how logistic regression can be defined in the makeLearner function.



```
# Define the model
learner <- makeLearner("classif.logreg", predict.type="prob", fix.factors.prediction=T)
```

Model Fitting

The model is trained with the training data. The below image shows that the training is carried out 'best_learner' which includes the best performing parameters and using the train task.

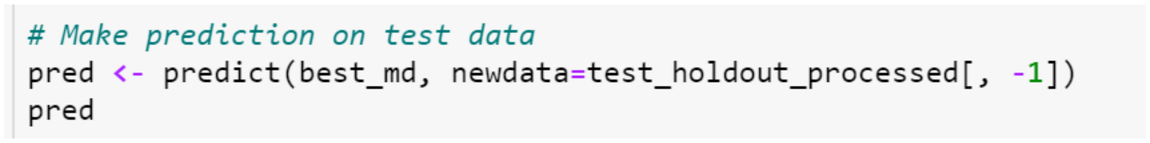


```
# Retrain the model with tbe best hyper-parameters
best_md <- mlr::train(best_learner, train_task)
```

The train_task given in the image allows the algorithm to learn about the target and the data.

train_task <- makeClassifTask(id="bank_train", data=train_processed[, -1], target="subscribe")

Once the training is done successfully the data can be used for prediction.



```
# Make prediction on test data
pred <- predict(best_md, newdata=test_holdout_processed[, -1])
pred
```

You can furthermore check the accuracy of the data to see the performance and do changes accordingly.

Pros of using the algorithm

1. Simple yet yield good results in many situations, hence is popular.
2. Ability to regularize the model to prevent overfitting.
3. Ability to yield probabilities as prediction results as well though it is a classification algorithm.
4. Allows easy regularization of outputs to prevent overfitting, yielding probabilities as prediction results.
5. Uses the stochastic gradient which allows optimal updating of the models.

Cons of using the algorithm

1. Fail to solve non-linear problems.
2. Fails to capture more complex relationships as it focuses on linear problems.
3. Logistic regression may underperform if the independent variables are not defined via Feature Engineering.
4. Underperforms when there are multiple decision boundaries.
5. Requires large sample sizes.

2. Decision Trees

Introduction

The decision tree is an intuitive model. The root indicates the inputs of the data while the leaves are the outcomes/ outputs. When using the decision trees the data is broken down in to parts by True or False answers about the variables. (Koehrsen, 2018) This would eventually lead to the final output of the dependent variable.

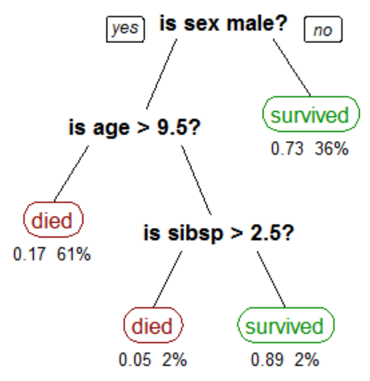


Figure 2: Source https://en.wikipedia.org/wiki/Decision_tree_learning

A decision tree is built by splitting the nodes by determining the questions which would reduce the amount of error in prediction. In other words the nodes are split in a way which brings the highest reduction in the Gini impurity. The length of branches indicates the amount of error reduced by the particular split. Higher length higher the reduction of error in prediction by the particular split.

Decision trees are known to be top down as it runs from the root and keeps on splitting till the final prediction. It is also known to be a Greedy Algorithm as it only focuses on the split part of the data at one point and does not consider the rest of the data. It is also binary as the data is split by half – True, False.

There is a risk of overfitting of data. Inorder to avoid overfitting pruning is carried out which can be either pre-pruning or post pruning.

Pros of using the algorithm

1. Simple
2. Easy to interpret
3. Does not require large amount of data but performs better with large amount of data
4. Able to handle both numerical and categorical data
5. Can be combined with other models.
6. A white box model as the explanation for the condition can be provided by a Boolean logic.

Cons of using the algorithm

1. Overfitting of the data is common.
2. High variance can occur when the model may also learn the noise which is presented in the data along with the other relationships.
3. Unstable as small change can lead to a large change in the structure of the decision tree.
4. Calculations can get complex if values are uncertain or linked. (Wikipedia, 2019)

3. Random Forest

Random Forest algorithm is a widely popular algorithm as well. It is made up of multiple decision trees. Hence each individual tree would function as explained in the 'Decision Trees' heading.

Random Forests use random sampling of training data points and random subsets of features when splitting the nodes.

In random sampling of training data points, the samples drawn with replacement is known as bootstrapping so that training on each tree is by different samples. Thereby though each tree may have high variance all trees together may have low variance.

The predictions are made by averaging each individual learners of the decision trees. This is known as bootstrap aggregating or bagging. (Koehrsen, 2018)

$$\hat{f} = \frac{1}{B}\sum_{b=1}^{B} f_b(x')$$

Thereby Random Forests overcomes the issue of overfitting of data carried out by Decision Trees.

When speaking of random subsets of features, Random Forests use a process known as 'Feature Bagging'. If one or few features seem to be strong predictors for multiple trees causing them to become correlated. For classification problems the features are rounded down so that the square root of the features are considered.

Pros of using the algorithm

1. Improved performance incomparison to Decision Trees
2. It has the ability to handle large number of variables and identify the most significant variables.
3. It can be considered as one of the dimensionality reduction method for its ability to select the significant variables in a given dataset.
4. It can solve both classification and regression problems.
5. It consists of bootstrap sampling where 1/3 of data which are not used for training shall be used for tasting.

Cons of using the algorithm

1. Random forest is a black box approach as the users have a lesser view and lesser control on what the Random Forest model uses for prediction.
2. It may overfit data which are noisy.

4. Gradient Boosting Machine

The concept of Gradient Boosting originated with the idea of optimizing algorithmns via the Cost function. It belongs to the category of iterative functional gradient descent algorithm. Gradient boosting would combine weak learners into creating a single strong learner in an iterative manner.

The model would predict values by minimizing the mean squared error. The algorithm would keep on improving Fm which is assumed as a weak model and adds the estimator h. Then to find h Gradient Boosting would initialize that with the perfect h. (h =y-Fm(x)

$$F_{m+1}(x) = F_m(x) + h(x) = y$$

or, equivalently,

$$h(x) = y - F_m(x).$$

Figure 3: Source - https://en.wikipedia.org/wiki/Gradient_boosting

The method tries to find the optimum value which minimizes the loss function on the training set, and incrementally expand it in a greedy fashion (that is focusing only on that particular scenario). Choosing the best function for h each step may be difficult or infeasible hence a steepest descent step is applied to support it (functional gradient descent).

$$F_m(x) = F_{m-1}(x) - \gamma_m \sum_{i=1}^{n} \nabla_{F_{m-1}} L(y_i, F_{m-1}(x_i)),$$

$$\gamma_m = \underset{\gamma}{\arg\min} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) - \gamma \nabla_{F_{m-1}} L(y_i, F_{m-1}(x_i))\right),$$

Figure 4: Updated model Source: https://en.wikipedia.org/wiki/Gradient_boosting

Pros of using the algorithm

1. Allows for high flexibility – options for hyper parameter tuning options.
2. Data preprocessing is not as required like in other models as it has the ability to handle both categorical and numerical variables.
3. Ability to be used for ranking such as in search engines
4. Handles missing data. (UC Business Analytics R Programming Guide, 2019)

Cons of using the algorithm

1. It takes high computation power and takes a long time to run.
2. GB is prone for overfitting.
3. It can be known as black box approach and will be hard to interpret.

5. XGBoost

XGBoost can be known as a fast gradient boosting frame work. XGBoost stands for 'Extreme Gradient Boosting. It was enhanced to improve the computation resources for boosted tree algorithms. (Brownlee, 2019) This is known to be an ensemble technique.

As previously explained in Gradient Boosting, Boosting is a method which allows weak learners to act together to form a strong learner.

It supports the three main types of gradient boosting.

1. Gradient Boosting.
2. Stochastic Gradient Boosting.
3. Regularized Gradient Boosting.

The algorithm has been enhanced to be able to support the parallerization of trees, automatic handling of missing data and continued training so that you can further boost your model. (Brownlee, 2019) Propotional shrinking of leaf nodes and extra randomization parameters are also available in XGBoost. (Wikipedia, 2020)

The XGBoost algorithm also functions similar to the Gradient Boosting algorithm.

The initial model Fm will be defined to predict y. The algorithm would add the estimator h. Then to find h Gradient Boosting would initialize that with the best h which is derived from (y-F0). By using the first model it would iterate to create new models until the residuals are minimized in the optimum way.

(Sundaram, 2018)

$$F_m(x) <- F_{m-1}(x) + h_m(x)$$

Figure 5: Source: https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/

Pros of using the algorithm

1. High execution speed. It supports parallel processing hence is faster than Gradient Boosting. Improved model performance.
2. XGBoost has built in regularization methods via Lasso Regression and Ridge regression.
3. Cross validation possible at each iteration.
4. Ability to handle missing values.
5. More effective tree pruning possible in comparison to Gradient Boosting.

Cons of using the algorithm

1. Despite the increased speed, training time can be high for large datasets.

2. Need to ensure data preparation is carried out prior to executing the model. Eg: Dummy variables for categorical variables.

6. QDA

QDA stands for Quadratic Discriminant Analysis. It is similar to LDA or Linear Discriminant Analysis. However as the names suggest LDA can only learn the linear boundaries while Quadratic Discriminant Analysis learn quadratic boundaries. It does not have hyperparameters to tune.

Both LDA and QDA derived from probabilistic models. It models the class conditional distribution of the data P(X|y=k) for each class k. Then the class K which maximizes this conditional probability is selected. P(X|y) is modeled as multivariate Gaussian distribution with density while d is the number of features. (scikit-learn developers, 2019)

$$P(X|y=k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(X-\mu_k)^t \Sigma_k^{-1} (X-\mu_k)\right)$$

Figure 6 Source: https://scikit-learn.org/stable/modules/lda_qda.html#id4

Pros of using the algorithm

1. Easy to compute.
2. Inherently multi class.
3. Fast classification.
4. Outperforms KNN and LDA

Cons of using the algorithm

1. Training time
2. Gaussian assumptions

Task 2: Setup the benchmark experiment to compare the 5 selected machine learning algorithms.

The data sets used to create models are a banking data set. The end objective is to predict whether a client is to subscribe a term deposit or not.

The data set contains bank client data, information of last contact and campaigns and social and economic context attributes. It contains both numerical and categorical data. The target variable itself is a categorical value.

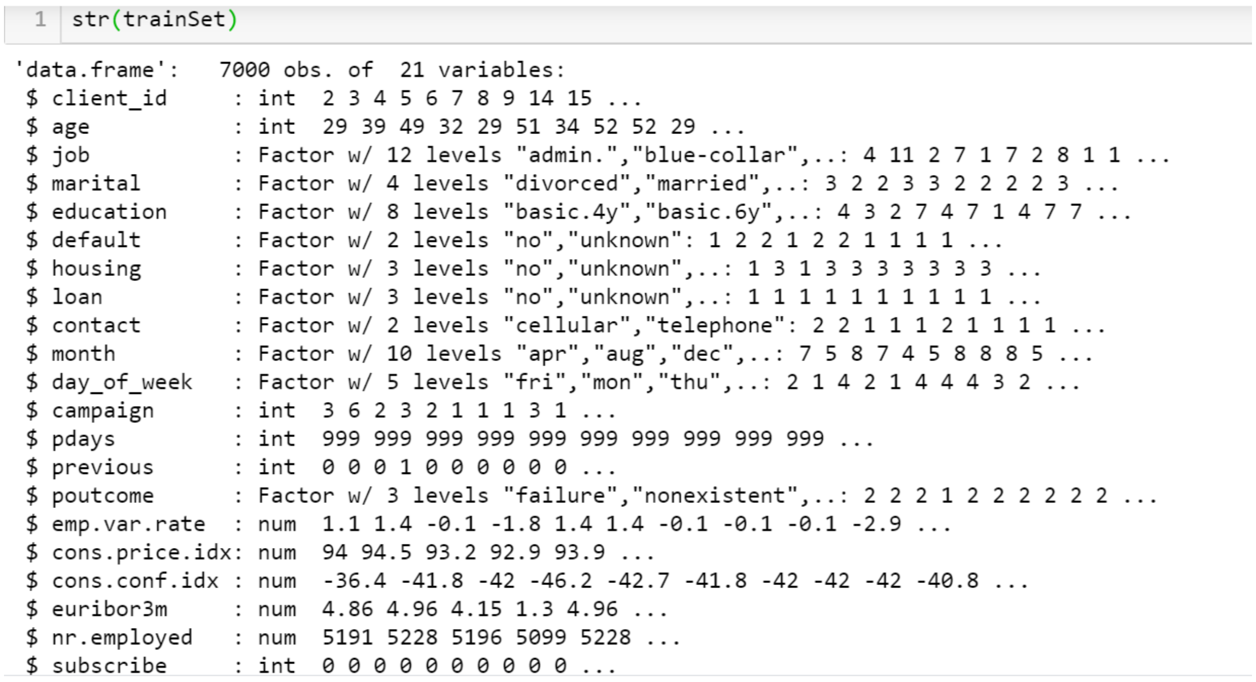
In order to start the experiment to compare the algorithms, all the relevant libraries are imported.

Then the relevant files are imported which are namely train set and the test hold out set which does not include the dependent variable which is the ‘subscribe’ column.

Investigate data

The data sets are explored to get an idea of the distribution of the data including the structure, number of missing values if any.

In the below data set you can view there are 7000 observations or rows with 21 variables or columns. It contains numerical values with decimals, integers and factors with character values as well.



```
str(trainSet)
```

```
'data.frame':	7000 obs. of  21 variables:
 $ client_id     : int  2 3 4 5 6 7 8 9 14 15 ...
 $ age           : int  29 39 49 32 29 51 34 52 52 29 ...
 $ job           : Factor w/ 12 levels "admin.","blue-collar",..: 4 11 2 7 1 7 2 8 1 1 ...
 $ marital       : Factor w/ 4 levels "divorced","married",..: 3 2 2 3 3 2 2 2 2 3 ...
 $ education     : Factor w/ 8 levels "basic.4y","basic.6y",..: 4 3 2 7 4 7 1 4 7 7 ...
 $ default       : Factor w/ 2 levels "no","unknown": 1 2 2 1 2 2 1 1 1 1 ...
 $ housing       : Factor w/ 3 levels "no","unknown",..: 1 3 1 3 3 3 3 3 3 3 ...
 $ loan          : Factor w/ 3 levels "no","unknown",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ contact       : Factor w/ 2 levels "cellular","telephone": 2 2 1 1 1 2 1 1 1 1 ...
 $ month         : Factor w/ 10 levels "apr","aug","dec",..: 7 5 8 7 4 5 8 8 8 5 ...
 $ day_of_week   : Factor w/ 5 levels "fri","mon","thu",..: 2 1 4 2 1 4 4 4 3 2 ...
 $ campaign      : int  3 6 2 3 2 1 1 1 3 1 ...
 $ pdays         : int  999 999 999 999 999 999 999 999 999 999 ...
 $ previous      : int  0 0 0 1 0 0 0 0 0 0 ...
 $ poutcome      : Factor w/ 3 levels "failure","nonexistent",..: 2 2 2 1 2 2 2 2 2 2 ...
 $ emp.var.rate  : num  1.1 1.4 -0.1 -1.8 1.4 1.4 -0.1 -0.1 -0.1 -2.9 ...
 $ cons.price.idx: num  94 94.5 93.2 92.9 93.9 ...
 $ cons.conf.idx : num  -36.4 -41.8 -42 -46.2 -42.7 -41.8 -42 -42 -42 -40.8 ...
 $ euribor3m     : num  4.86 4.96 4.15 1.3 4.96 ...
 $ nr.employed   : num  5191 5228 5196 5099 5228 ...
 $ subscribe     : int  0 0 0 0 0 0 0 0 0 0 ...
```

To visualize the distribution of certain parameters you can also proceed to view variables using graphical formats.

When checking for missing values it was found that there were no missing values. You can also proceed to correct any values if there are any, at this stage.

Split the data

Next, you need to split the original training data as train and test. The train and test data is split by 75% and 25% accordingly. You can also proceed splitting the data as train, validation and test.

In this project, I have split the data as train and test providing more data for training and testing accordingly.

You can view the percentage distribution of the target variable values and ensure there is no significant difference percentage-wise in order to continue.

Feature Engineering

Create new variables

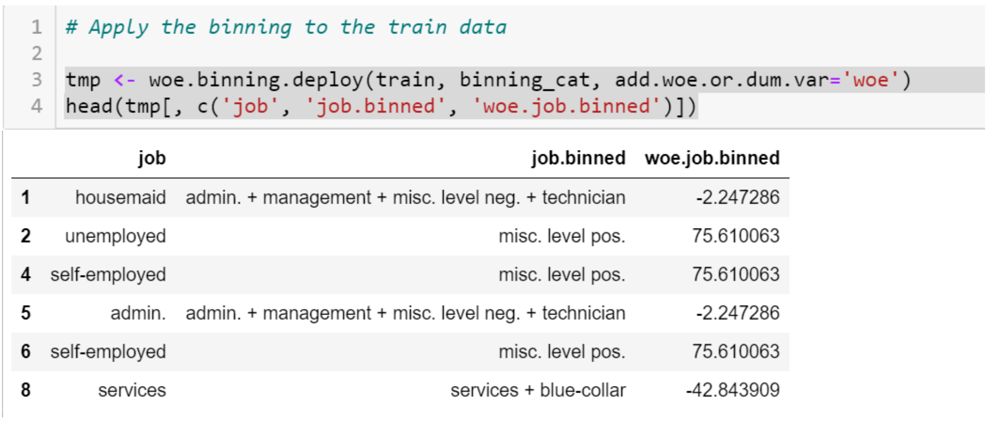
Once the data is investigated and corrected appropriately, feature engineering process can be carried out. The below shows an example how months are broken down to seasons and assigned to appropriate columns.

Also note that all the newly created features / variables need to be added to all data sets, that is Train, Test and the final Predict set (test hold out set).

```
# Add new seasons variable to train and test (holdout)
# Train
train[, 'month_spring'] <- as.integer(train$month %in% c('mar','apr', 'may'))
train[, 'month_summer'] <- as.integer(train$month %in% c('jun','jul', 'aug'))
train[, 'month_autumn'] <- as.integer(train$month %in% c('sep','oct', 'nov'))
train[, 'month_winter'] <- as.integer(train$month %in% c('dec','jan', 'feb'))

# Test
test[, 'month_spring'] <- as.integer(test$month %in% c('mar','apr', 'may'))
test[, 'month_summer'] <- as.integer(test$month %in% c('jun','jul', 'aug'))
test[, 'month_autumn'] <- as.integer(test$month %in% c('sep','oct', 'nov'))
test[, 'month_winter'] <- as.integer(test$month %in% c('dec','jan', 'feb'))


#PredictSet
predictSet[, 'month_spring'] <- as.integer(predictSet$month %in% c('mar','apr', 'may'))
predictSet[, 'month_summer'] <- as.integer(predictSet$month %in% c('jun','jul', 'aug'))
predictSet[, 'month_autumn'] <- as.integer(predictSet$month %in% c('sep','oct', 'nov'))
predictSet[, 'month_winter'] <- as.integer(predictSet$month %in% c('dec','jan', 'feb'))
```

Create bins/groups

In the feature engineering process you can also proceed to create groups based on the variables and the effect on the target variable. Below shows how jobs were grouped or binned using the ‘woe.binning.deploy’ function in woeBinning package. You can create bins on character as well as numerical variables.



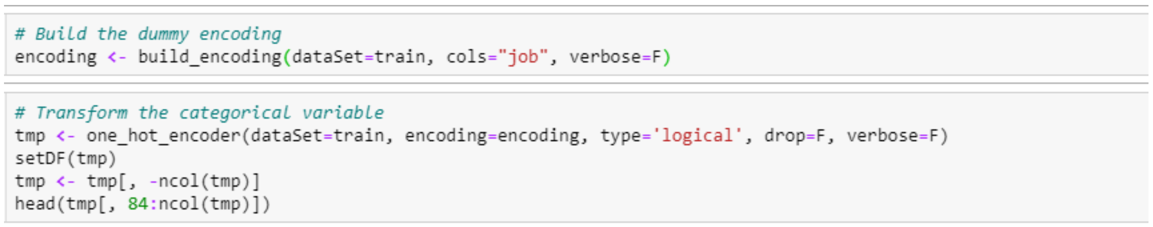
```r
# Apply the binning to the train data

tmp <- woe.binning.deploy(train, binning_cat, add.woe.or.dum.var='woe')
head(tmp[, c('job', 'job.binned', 'woe.job.binned')])
```

	job	job.binned	woe.job.binned
1	housemaid	admin. + management + misc. level neg. + technician	-2.247286
2	unemployed	misc. level pos.	75.610063
4	self-employed	misc. level pos.	75.610063
5	admin.	admin. + management + misc. level neg. + technician	-2.247286
6	self-employed	misc. level pos.	75.610063
8	services	services + blue-collar	-42.843909

You need to make sure that the created variables as assigned on your datasets as before.

Converting Categorical to readable formats

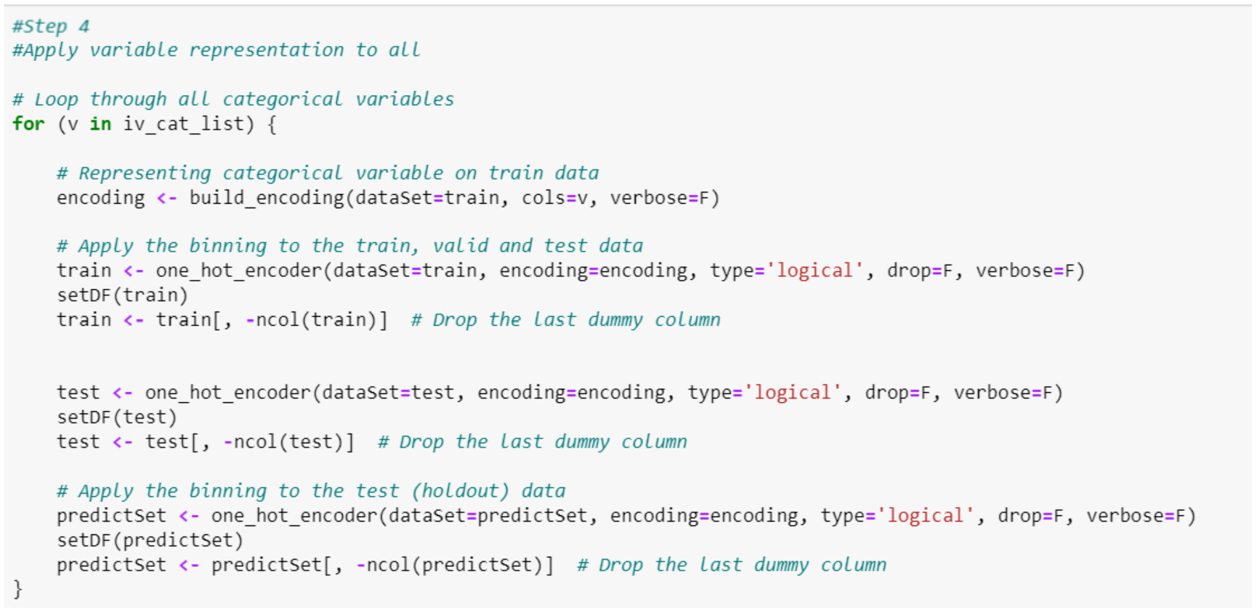
Once all the new features are created you could proceed to convert categorical to dummy or numerical variables which is easily readable by the algorithms.



```r
# Build the dummy encoding
encoding <- build_encoding(dataSet=train, cols="job", verbose=F)
```

```r
# Transform the categorical variable
tmp <- one_hot_encoder(dataSet=train, encoding=encoding, type='logical', drop=F, verbose=F)
setDF(tmp)
tmp <- tmp[, -ncol(tmp)]
head(tmp[, 84:ncol(tmp)])
```

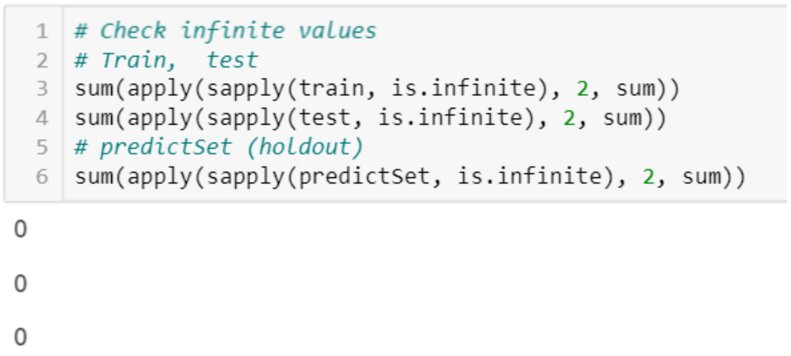
Ensure to include the created variables in all three datasets



```r
#Step 4
#Apply variable representation to all

# Loop through all categorical variables
for (v in iv_cat_list) {

    # Representing categorical variable on train data
    encoding <- build_encoding(dataSet=train, cols=v, verbose=F)

    # Apply the binning to the train, valid and test data
    train <- one_hot_encoder(dataSet=train, encoding=encoding, type='logical', drop=F, verbose=F)
    setDF(train)
    train <- train[, -ncol(train)]  # Drop the last dummy column


    test <- one_hot_encoder(dataSet=test, encoding=encoding, type='logical', drop=F, verbose=F)
    setDF(test)
    test <- test[, -ncol(test)]  # Drop the last dummy column

    # Apply the binning to the test (holdout) data
    predictSet <- one_hot_encoder(dataSet=predictSet, encoding=encoding, type='logical', drop=F, verbose=F)
    setDF(predictSet)
    predictSet <- predictSet[, -ncol(predictSet)]  # Drop the last dummy column
}
```

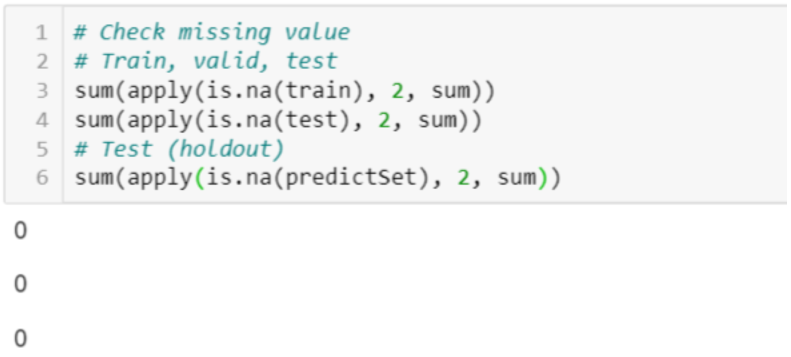
You can furthermore convert to standardize the numerical values for better performance and accuracy of the algorithms.

Prior to proceeding the next step, check for missing values and infinite values in your data set.



```r
# Check infinite values
# Train,  test
sum(apply(sapply(train, is.infinite), 2, sum))
sum(apply(sapply(test, is.infinite), 2, sum))
# predictSet (holdout)
sum(apply(sapply(predictSet, is.infinite), 2, sum))
```

```
0

0

0
```

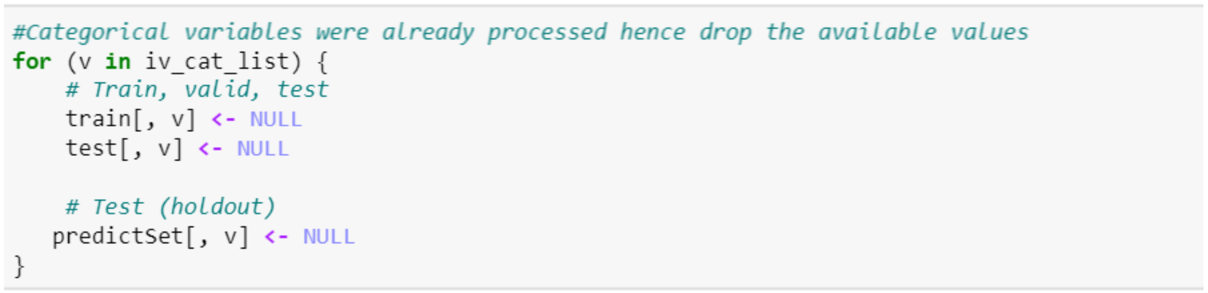


```r
# Check missing value
# Train, valid, test
sum(apply(is.na(train), 2, sum))
sum(apply(is.na(test), 2, sum))
# Test (holdout)
sum(apply(is.na(predictSet), 2, sum))
```

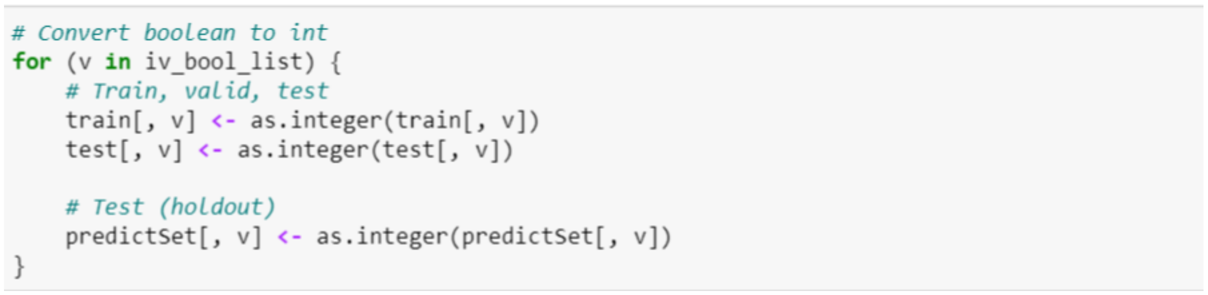
```
0

0

0
```

If you find infinite or missing values ensure to treat them or drop them accordingly.

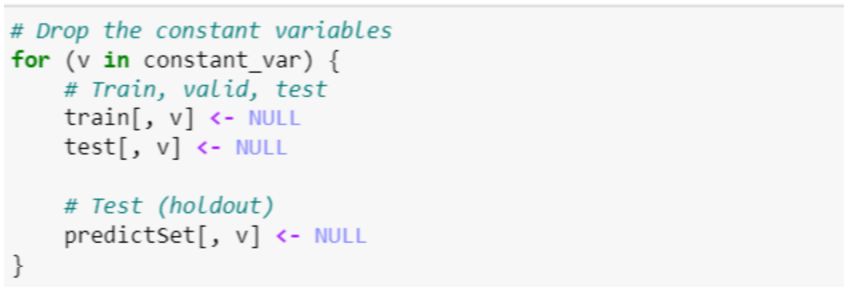
Other than that after you have created dummy variables representing the categorical variable drop the categorical variables and constant variables. You also need to ensure to convert the Boolean values to integers as well.



```
#Categorical variables were already processed hence drop the available values
for (v in iv_cat_list) {
    # Train, valid, test
    train[, v] <- NULL
    test[, v] <- NULL

    # Test (holdout)
   predictSet[, v] <- NULL
}
```



```
# Convert boolean to int
for (v in iv_bool_list) {
    # Train, valid, test
    train[, v] <- as.integer(train[, v])
    test[, v] <- as.integer(test[, v])

    # Test (holdout)
    predictSet[, v] <- as.integer(predictSet[, v])
}
```



```
# Drop the constant variables
for (v in constant_var) {
    # Train, valid, test
    train[, v] <- NULL
    test[, v] <- NULL

    # Test (holdout)
    predictSet[, v] <- NULL
}
```

Feature Selection

There are variables ways to carry out feature selection. Here we have used the Fisher Score. It would provide a scoring to the best independent variables and select the best variables from the score. This would also help to reduce noise in the algorithms.

Feature Selection

```r
FisherScore <- function(basetable, depvar, IV_list) {
  "
  This function calculate the Fisher score of a variable.

  Ref:
  ---
  Verbeke, W., Dejaeger, K., Martens, D., Hur, J., & Baesens, B. (2012). New insights into churn prediction in the telecommu
  "

  # Get the unique values of dependent variable
  DV <- unique(basetable[, depvar])

  IV_FisherScore <- c()

  for (v in IV_list) {
    fs <- abs((mean(basetable[which(basetable[, depvar]==DV[1]), v]) - mean(basetable[which(basetable[, depvar]==DV[2]), v])
      sqrt((var(basetable[which(basetable[, depvar]==DV[1]), v]) + var(basetable[which(basetable[, depvar]==DV[2]), v])))
    IV_FisherScore <- c(IV_FisherScore, fs)
  }

  return(data.frame(IV=IV_list, fisher_score=IV_FisherScore))
}

varSelectionFisher <- function(basetable, depvar, IV_list, num_select=20) {
  "
  This function will calculate the Fisher score for all IVs and select the best
  top IVs.

  Assumption: all variables of input dataset are converted into numeric type.
  "

  fs <- FisherScore(basetable, depvar, IV_list)  # Calculate Fisher Score for all IVs
  num_select <- min(num_select, ncol(basetable))  # Top N IVs to be selected
  return(as.vector(fs[order(fs$fisher_score, decreasing=T), ][1:num_select, 'IV']))
}
```

In this project I have selected the top 30 features to be used for prediction.

```r
# Select top 30 variables according to the Fisher Score
best_fs_var <- varSelectionFisher(train, dv_list, iv_list, num_select=30)
head(best_fs_var,30)
```

```
'woe.euribor3m.binned'   'euribor3m'   'euribor3m.binned_incidence'   'emp.var.rate'   'nr.employed'   'nr.employed.binned...Inf.0.'
'nr.employed.binned..0. Inf.'   'nr.employed.binned_incidence'   'woe.nr.employed.binned'   'euribor_ge_mean'   'woe.euribor_ge_mean.binned'
'euribor.ge.mean.binned...Inf.0.'   'euribor.ge.mean.binned..0. Inf.'   'euribor_ge_mean.binned_incidence'   'woe.emp.var.rate.binned'
'euribor3m.binned...Inf.1.262.'   'emp.var.rate.binned_incidence'   'emp.var.rate.binned...0.1. Inf.'   'euribor3m.binned..4.153. Inf.'   'woe.cons.conf.idx.binned'
'cons.conf.idx.binned_incidence'   'woe.cons.price.idx.binned'   'cons.price.idx.binned_incidence'   'month_incidence'   'pdays'   'pdays_notcontacted'
'woe.month.binned'   'woe.poutcome.binned'   'month.binned_incidence'   'poutcome_incidence'
```

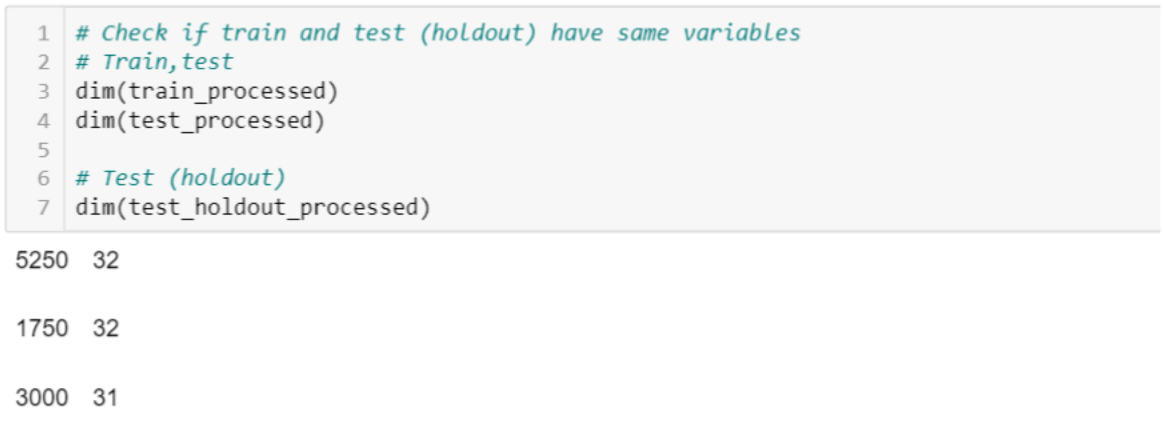
Once the features are selected accordingly, ensure that the selected features are reflected on the three data sets as provided below. Thereby only the top 30 features are retained in the algorithm.

```r
# Apply variable selection to the data
# Train
var_select <- names(train)[names(train) %in% best_fs_var]
train_processed <- train[, c('client_id', var_select, 'subscribe')]

# Test
var_select <- names(test)[names(test) %in% best_fs_var]
test_processed <- test[, c('client_id', var_select, 'subscribe')]
# Test (holdout)
var_select <- names(predictSet)[names(predictSet) %in% best_fs_var]
test_holdout_processed <- predictSet[, c('client_id', var_select)]
```

Check train and test sets have the same columns

Before proceeding to the next step ensure that the train test and test holdout data sets have the same columns or variables. It can be viewed by the dim function.

The below displays that all three data sets have been assigned with the selected features accordingly inaddition to the client_id and subscribe. (Note the test_holdout dataset does not have a ‘subscribe’ column hence it would have 31 columns in total)

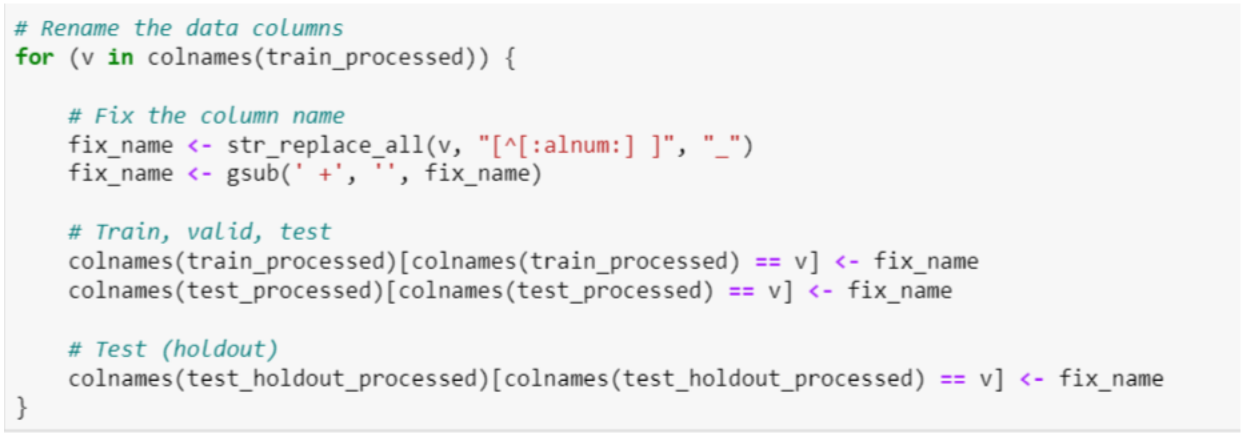


```
# Check if train and test (holdout) have same variables
# Train,test
dim(train_processed)
dim(test_processed)

# Test (holdout)
dim(test_holdout_processed)
```

```
5250  32

1750  32

3000  31
```

Finally, you should fix the column names in the base tables as this would create errors in the algorithms when processing.

The below code replaces symbols and spaces with ‘_’ in all three applicable datasets.



```
# Rename the data columns
for (v in colnames(train_processed)) {

    # Fix the column name
    fix_name <- str_replace_all(v, "[^[:alnum:] ]", "_")
    fix_name <- gsub(' +', '', fix_name)

    # Train, valid, test
    colnames(train_processed)[colnames(train_processed) == v] <- fix_name
    colnames(test_processed)[colnames(test_processed) == v] <- fix_name

    # Test (holdout)
    colnames(test_holdout_processed)[colnames(test_holdout_processed) == v] <- fix_name
}
```

Model the algorithms

The algorithms are built mainly using the MLR package. This package allows to implement various algorithms using same set of commands.

We have created several algorithms namely,

1. Logistic Regression
2. Random Forest
3. Decision Trees
4. Gradient Boosting
5. XGBoost
6. QDA

For the reason that this is a benchmark experiment, the setups are kept as similar as possible. To reduce the repetitive nature, the set up of Logistic Regression would be explained in detail. Thereby only the additional changes for other models would be explained under the headings of each of those models.

1. Logistic Regression

The Logistic regression was built on the final data set : **train_processed** which was created.

Initially when building the model we define the model as **classif.logreg** for logistic regression for classification. We have further defined additional properties as predict type '**prob**' and fix factors prediction to true. Prob indicates it would select items with the maximal probabilities.By defining and fix factors prediction as true, it would allow learner to repair any issues with factors for new data sets.

We also set up cross validation in order to improve accuracy. This is set up by **makeResampleDesc**. The i**ters** number indicates that it is a five fold cross validation. (In this project I have used 5 fold cross validation in all models.) Cross validation would help to evaluate the machine learning model.It would shuffle the data randomly and split the data in to the number groups (K) and there by in each unique group it would create one part as a test hold out and the remaining part would be used as training data set. It would repetitively carry out that task for all the models. Then it would evaluate based on all the scores of the groups.

Next, we define the classification task for the model to perform on where the target variable is defined.

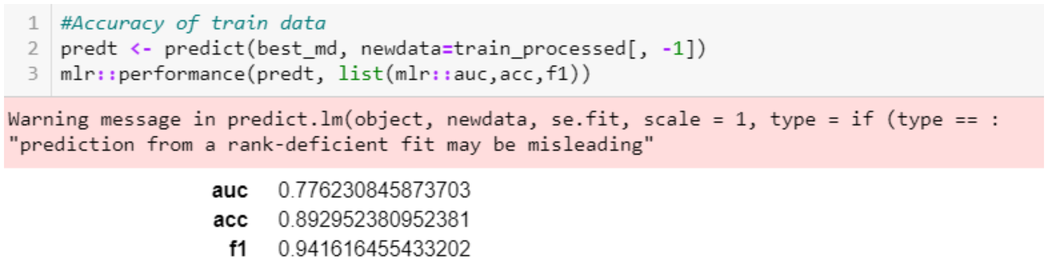
The hyper parameters are set. Define a tune control grid. Next the hyper parameter tuning is run with the 5-fold cross validation which we previously defined. It would then pick the best hyper parameters for the learning algorithm which would have the highest auc in the model.

```r
# Set up cross-validation
rdesc = makeResampleDesc("CV", iters=5, predict="both")

# Define the model
learner <- makeLearner("classif.logreg", predict.type="prob", fix.factors.prediction=T)

# Define the task
train_task <- makeClassifTask(id="bank_train", data=train_processed[, -1], target="subscribe")

# Set hyper parameter tuning
tune_params <- makeParamSet(
)
ctrl = makeTuneControlGrid()

# Run the hyper parameter tuning with k-fold CV
if (length(tune_params$pars) > 0) {
    # Run parameter tuning
    res <- tuneParams(learner, task=train_task, resampling=rdesc,
      par.set=tune_params, control=ctrl, measures=list(mlr::auc))

    # Extract best model
    best_learner <- res$learner

} else {
    # Simple cross-validation
    res <- resample(learner, train_task, rdesc, measures=list(mlr::auc, setAggregation(mlr::auc, train.mean)))

    # No parameter for tuning, only 1 best learner
    best_learner <- learner
}
```

When this model is run it would train with the train_processed data set and would pick the best parameters associated to derive the highest AUC. The resampling is done with the 5 fold cross validation.

Next, we need to retrain the model with the previously selected best hyper parameters.

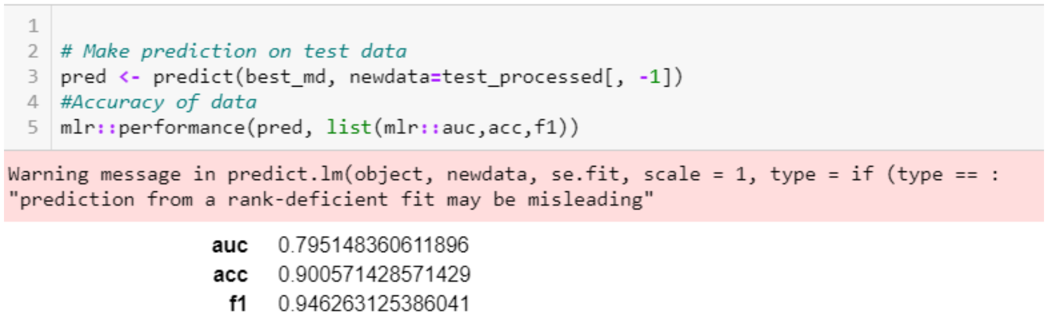
Finally we check the accuracies of both train_processed and test_ processed data sets using AUC, Accuracy and FI metrics.



```
#Accuracy of train data
predt <- predict(best_md, newdata=train_processed[, -1])
mlr::performance(predt, list(mlr::auc,acc,f1))
```

```
Warning message in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
"prediction from a rank-deficient fit may be misleading"
```

```
auc   0.776230845873703
acc   0.892952380952381
 f1   0.941616455433202
```



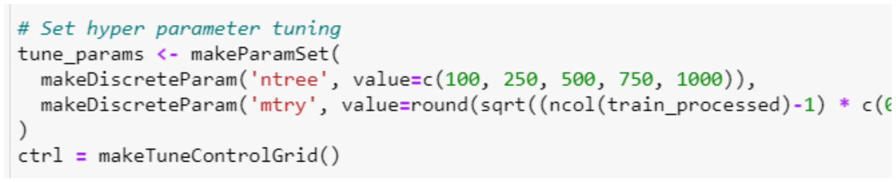
```

# Make prediction on test data
pred <- predict(best_md, newdata=test_processed[, -1])
#Accuracy of data
mlr::performance(pred, list(mlr::auc,acc,f1))
```

```
Warning message in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
"prediction from a rank-deficient fit may be misleading"
```

```
auc   0.795148360611896
acc   0.900571428571429
 f1   0.946263125386041
```

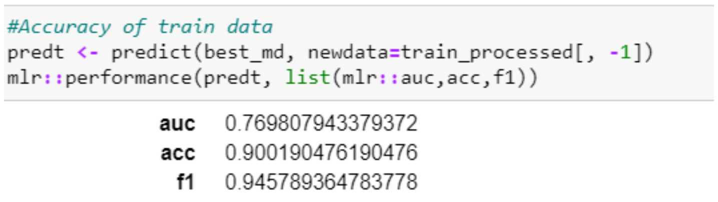
2. Random Forest

In the random forest most of the set is similar to Logistic Regression model. We define model as Random Forest. However we are able to define number of trees and number of try as the hyper parameters.



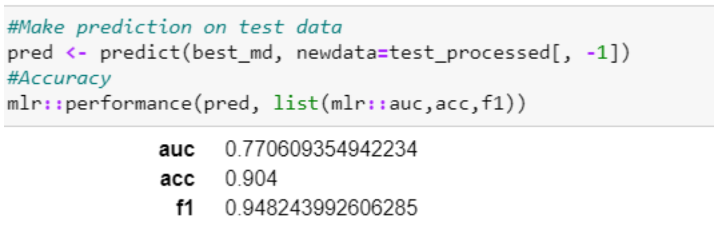
```
# Set hyper parameter tuning
tune_params <- makeParamSet(
  makeDiscreteParam('ntree', value=c(100, 250, 500, 750, 1000)),
  makeDiscreteParam('mtry', value=round(sqrt((ncol(train_processed)-1) * c(
)
ctrl = makeTuneControlGrid()
```

Similar to the above set up we have calculated the AUC, ACC and FI of the models.



```
#Accuracy of train data
predt <- predict(best_md, newdata=train_processed[, -1])
mlr::performance(predt, list(mlr::auc,acc,f1))
```

```
auc   0.769807943379372
acc   0.900190476190476
 f1   0.945789364783778
```

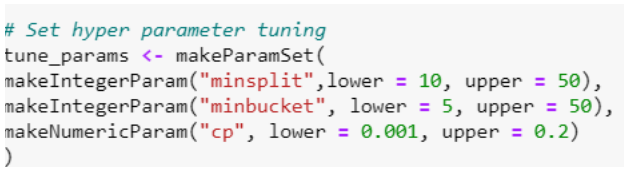


```
#Make prediction on test data
pred <- predict(best_md, newdata=test_processed[, -1])
#Accuracy
mlr::performance(pred, list(mlr::auc,acc,f1))
```

```
auc   0.770609354942234
acc   0.904
 f1   0.948243992606285
```

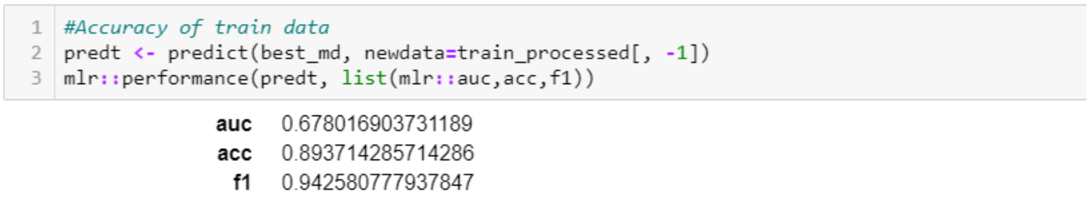
3. Decision Trees

In the Decision Tree model most of the set up is similar to Random Forest model. We define model as **classif.rpart** in the MLR function makeLearner. However we are able to define different parameters such as minsplit, min bucket and cp.



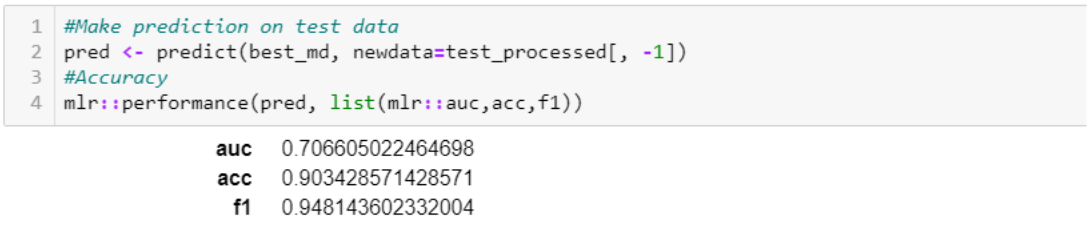
```
# Set hyper parameter tuning
tune_params <- makeParamSet(
makeIntegerParam("minsplit",lower = 10, upper = 50),
makeIntegerParam("minbucket", lower = 5, upper = 50),
makeNumericParam("cp", lower = 0.001, upper = 0.2)
)
```

Similar to the above set up we have calculated the AUC, ACC and FI of the models so we could compare with all the models.



```
#Accuracy of train data
predt <- predict(best_md, newdata=train_processed[, -1])
mlr::performance(predt, list(mlr::auc,acc,f1))
```

```
auc  0.678016903731189
acc  0.893714285714286
 f1  0.942580777937847
```



```
#Make prediction on test data
pred <- predict(best_md, newdata=test_processed[, -1])
#Accuracy
mlr::performance(pred, list(mlr::auc,acc,f1))
```

```
auc  0.706605022464698
acc  0.903428571428571
 f1  0.948143602332004
```

4. Gradient Boosting

In the Gradient Boosting model most of the set up is similar to the above models. We define model as **classif.gbm** in the MLR function makeLearner. However we are able to define different parameters for this model as well.

I have defined the distribution as "Bernoulli" and defined lower and upper limits for the number of trees. The depth of the trees were also defined in the parameters.

Shrinkage in Gradient Boosted trees allows regularization. Smaller learning rates have good improvements of the model yet the computation time was higher with regards to training and querying.

```
# Set hyper parameter tuning
tune_params <-makeParamSet(
makeDiscreteParam("distribution", values = "bernoulli"),
makeIntegerParam("n.trees", lower = 100, upper = 1000), #number of trees
makeIntegerParam("interaction.depth", lower = 2, upper = 10), #depth of tree
makeNumericParam("shrinkage",lower = 0.01, upper = 1)
)
ctrl = makeTuneControlGrid()
```

Similar to the above set up we have calculated the AUC, ACC and FI of the models so we could compare with all the models.

```
#Accuracy of train data
predt <- predict(best_md, newdata=train_processed[, -1])
mlr::performance(predt, list(mlr::auc,acc,f1))
```

```
auc   0.788011921940493
acc   0.895238095238095
 f1   0.94300518134715
```

```
#Make prediction on test data
pred <- predict(best_md, newdata=test_processed[, -1])
#Accuracy
mlr::performance(pred, list(mlr::auc,acc,f1))
```

```
auc   0.801670477642276
acc   0.906285714285714
 f1   0.949476278496611
```

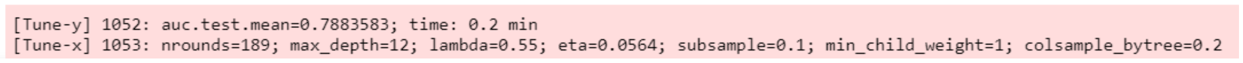
5. XGBoost

XGBoost is also a Boosting model and most of the set up is similar to the above models. We define model as **classif.xgboost** in the MLR function makeLearner. However we are able to define different parameters for this model as well.

```
# Set hyper parameter tuning
tune_params <-makeParamSet(
makeIntegerParam("nrounds",lower=100,upper=500),
makeIntegerParam("max_depth",lower=3,upper=20),
makeNumericParam("lambda",lower=0.55,upper=0.60),
makeNumericParam("eta", lower = 0.01, upper = 0.5),
makeNumericParam("subsample", lower = 0.30, upper = 0.80),
makeNumericParam("min_child_weight",lower=1,upper=5),
makeNumericParam("colsample_bytree",lower = 0.3,upper = 0.8)
)
ctrl = makeTuneControlGrid()
```

Although this was to perform faster than the Gradient Boosting function, this took much longer to process as I did some changes. Previously also this when it comes to this model the accuracy was lower than the other models.

However, though the calculation process is still carried out the auc mean was given as 0.788



```
[Tune-y] 1052: auc.test.mean=0.7883583; time: 0.2 min
[Tune-x] 1053: nrounds=189; max_depth=12; lambda=0.55; eta=0.0564; subsample=0.1; min_child_weight=1; colsample_bytree=0.2
```

Evaluation

The evaluation metrics used are AUC, Accuracy and F1.

- Accuracy – Accuracy is calculated using True/ True+False. It would provide the basic performance of the model.
- AUC – AUC is a better metric than accuracy as accuracy may tend to give false sense of high accuracy levels in various situations. AUC is the area under ROC curve and uses a confusion matrix to derive the calculation. It is one of the commonly accepted metrics for evaluating algorithms.
- F1 – F1 score is calculated using the weighted average of Precision and Recall. The F1 score is also used in the case where Accuracy or AUC may get affected by any skew in data (false positive being higher than true positive etc).

As per the data available based on the evaluations of the above mentioned metrics Gradient Boosting Tree performs better than all the models with an AUC score of 0.802, Accuracy of 0.906 and an F1 Score of 0.946 .

Logistic Regression follows up with an AUC of 0.795 and an F1 score of 0.946. Random Forest closely follows up with a slightly better F1 score well as Accuracy than Logistic Regression. However, the AUC is comparatively much lower than the Logistic Regression.

Note: XGBoost performed lower than Gradient Boosting in a previous run. Yet, I was unable to get the latest scores as it took more than 6 hours to run though it gave a mean AUC score of 0.788 as per the running statistics.

Model	**Train data set**				**Test data set**		
	AUC	Accuracy	F1		AUC	Accuracy	F1
Logistic Regression	0.776	0.892	0.941		0.795	0.901	0.946
Random Forest	0.770	0.900	0.946		0.771	0.904	0.948
Decision Trees	0.678	0.894	0.943		0.707	0.903	0.948
Gradient Boosting	0.788	0.895	0.943		0.802	0.906	0.949
XGBoost	0.788						

References

Brownlee, J. (2019). *A Gentle Introduction to XGBoost for Applied Machine Learning*. Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/

Chatterjee, S. (2019). *A Comprehensive Study of Linear vs Logistic Regression to refresh the Basics*. Retrieved from Medium: https://towardsdatascience.com/a-comprehensive-study-of-linear-vs-logistic-regression-to-refresh-the-basics-7e526c1d3ebe

Koehrsen, W. (2018). *An Implementation and Explanation of the Random Forest in Python*. Retrieved from Towards Data Science: https://towardsdatascience.com/an-implementation-and-explanation-of-the-random-forest-in-python-77bf308a9b76

Li, S. (2017). *Building A Logistic Regression in Python, Step by Step*. Retrieved from Medium: https://towardsdatascience.com/building-a-logistic-regression-in-python-step-by-step-becd4d56c9c8

Prabhakaran, S. (2017). *Logistic Regression – A Complete Tutorial With Examples in R*. Retrieved from Machine Learning Plus: https://www.machinelearningplus.com/machine-learning/logistic-regression-tutorial-examples-r/

scikit-learn developers. (2019). *1.2. Linear and Quadratic Discriminant Analysis*. Retrieved from scikit learn: https://scikit-learn.org/stable/modules/lda_qda.html#id4

Sundaram, R. B. (2018). *An End-to-End Guide to Understand the Math behind XGBoost*. Retrieved from Analytics Vidya: https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/

UC Business Analytics R Programming Guide. (2019). *Gradient Boosting Machines*. Retrieved from UC Business Analytics R Programming Guide: http://uc-r.github.io/gbm_regression

Wikipedia. (2019). *Decision Tree*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Decision_tree

Wikipedia. (2020). *XGBoost*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/XGBoost#Features