Loan Approval Classification

David Apamo

2024-11-13

# Introduction

In the highly competitive and risk-laden field of finance, ensuring the reliability of loan approvals is paramount for banks and financial institutions. The proliferation of bad loans can significantly impact the financial stability and profitability of these institutions. To address this challenge, I propose the development of a robust classification model aimed at predicting loan approval outcomes with high accuracy. By leveraging advanced machine learning techniques, this study will analyze a variety of borrower attributes and historical loan data to distinguish between high-risk and low-risk loan applicants. The objective is to provide a decision-making tool that enhances the precision of loan approvals, thereby mitigating financial losses and fostering sustainable financial growth. This initiative underscores the vital role of data-driven solutions in modern finance, promising significant improvements in risk management and operational efficiency.

# Load packages  
suppressMessages(  
 {  
 library(tidyverse)  
 library(janitor)  
 library(caret)  
 library(mlr)  
 library(tidymodels)  
 library(pROC)  
 library(vip)  
 library(corrplot)  
 library(parallel)  
 library(parallelMap)  
 }  
)

# Import data  
Loan\_data <- read.csv("Loan\_Data.csv")

# View the structure of the data  
Loan\_data |> str()

## 'data.frame': 614 obs. of 13 variables:  
## $ Loan\_ID : chr "LP001002" "LP001003" "LP001005" "LP001006" ...  
## $ Gender : chr "Male" "Male" "Male" "Male" ...  
## $ Married : chr "No" "Yes" "Yes" "Yes" ...  
## $ Dependents : chr "0" "1" "0" "0" ...  
## $ Education : chr "Graduate" "Graduate" "Graduate" "Not Graduate" ...  
## $ Self\_Employed : chr "No" "No" "Yes" "No" ...  
## $ ApplicantIncome : int 5849 4583 3000 2583 6000 5417 2333 3036 4006 12841 ...  
## $ CoapplicantIncome: num 0 1508 0 2358 0 ...  
## $ LoanAmount : int NA 128 66 120 141 267 95 158 168 349 ...  
## $ Loan\_Amount\_Term : int 360 360 360 360 360 360 360 360 360 360 ...  
## $ Credit\_History : int 1 1 1 1 1 1 1 0 1 1 ...  
## $ Property\_Area : chr "Urban" "Rural" "Urban" "Urban" ...  
## $ Loan\_Status : chr "Y" "N" "Y" "Y" ...

The data has 614 observations of 13 variables. The variables applicant income, co-applicant income, loan amount, loan amount term and credit history are numeric while the rest of the variables are character.

# View the first six observations  
Loan\_data |> head()

## Loan\_ID Gender Married Dependents Education Self\_Employed ApplicantIncome  
## 1 LP001002 Male No 0 Graduate No 5849  
## 2 LP001003 Male Yes 1 Graduate No 4583  
## 3 LP001005 Male Yes 0 Graduate Yes 3000  
## 4 LP001006 Male Yes 0 Not Graduate No 2583  
## 5 LP001008 Male No 0 Graduate No 6000  
## 6 LP001011 Male Yes 2 Graduate Yes 5417  
## CoapplicantIncome LoanAmount Loan\_Amount\_Term Credit\_History Property\_Area  
## 1 0 NA 360 1 Urban  
## 2 1508 128 360 1 Rural  
## 3 0 66 360 1 Urban  
## 4 2358 120 360 1 Urban  
## 5 0 141 360 1 Urban  
## 6 4196 267 360 1 Urban  
## Loan\_Status  
## 1 Y  
## 2 N  
## 3 Y  
## 4 Y  
## 5 Y  
## 6 Y

## Data Cleaning and Preprocessing

# Check for missing values  
map\_dbl(Loan\_data, ~sum(is.na(.)))

## Loan\_ID Gender Married Dependents   
## 0 0 0 0   
## Education Self\_Employed ApplicantIncome CoapplicantIncome   
## 0 0 0 0   
## LoanAmount Loan\_Amount\_Term Credit\_History Property\_Area   
## 22 14 50 0   
## Loan\_Status   
## 0

Only missing values of the numeric features are shown. Loan amount has 22 missing values, loan amount term has 14 missing values while credit history has 50 missing values. The missing values for the character variables aren’t recorded as NA hence were not detected.

# Fill in the white spaces in character variables with NA  
Loan\_data <- Loan\_data |> mutate\_if(is.character, ~ na\_if(., ""))

# Recount the number of missing values in each column  
map\_dbl(Loan\_data, ~sum(is.na(.)))

## Loan\_ID Gender Married Dependents   
## 0 13 3 15   
## Education Self\_Employed ApplicantIncome CoapplicantIncome   
## 0 32 0 0   
## LoanAmount Loan\_Amount\_Term Credit\_History Property\_Area   
## 22 14 50 0   
## Loan\_Status   
## 0

All the missing values are now detected. Gender has 13, married has 3, dependents has 15, self employed has 32, loan amount has 22, loan amount term has 14 and credit history has 50 missing values respectively.

# Check for duplicated observations  
sum(duplicated(Loan\_data))

## [1] 0

There are no duplicated observations in the data.

# Clean variable names by converting them all to lowercase  
Loan\_data <- Loan\_data |> clean\_names()

# Convert the character variables to factors  
  
# Specify columns to factor  
cols\_to\_factor <- c("gender", "married", "dependents", "education",  
 "self\_employed", "property\_area")  
  
# Convert the specified columns into factors  
Loan\_data <- Loan\_data |> mutate\_at(.vars = cols\_to\_factor, .fun = factor)

# Factor credit history variable  
Loan\_data$credit\_history <- factor(Loan\_data$credit\_history,   
 labels = c("Bad","Good"),   
 levels = c(0,1))

# Convert loan status variable into a binary variable  
Loan\_data$loan\_status <- ifelse(Loan\_data$loan\_status == "Y", 1, 0)  
  
# Factor the variable and reverse the order of the labels to begin with the positive class (This is important for model training)  
  
Loan\_data$loan\_status <- factor(Loan\_data$loan\_status,   
 levels = rev(c(0,1)),   
 labels = rev(c("Not Approved", "Approved")))

# Missing Value Imputation

I’ll use KNN imputation method because it isn’t easy to tell the nature of missingness, whether the values are missing not at random, missing at random or missing completely at random.

# Load the VIM package for missing value imputation  
suppressMessages(library(VIM))  
# Impute based on 5 nearest data points  
df\_imputed <- kNN(Loan\_data, variable = colnames(Loan\_data),   
 k = 5)

# Subset the complete data with imputed values (omit loan ID and the logical variables indicating whether a variables has been imputed or not)  
Data <- df\_imputed[, 2:13]

# EDA

# Generate statistical summary for each and every variable  
summary(Data)

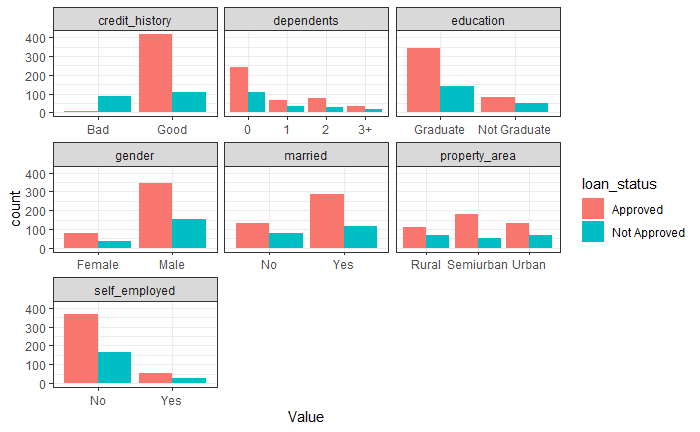
## gender married dependents education self\_employed  
## Female:113 No :213 0 :354 Graduate :480 No :532   
## Male :501 Yes:401 1 :103 Not Graduate:134 Yes: 82   
## 2 :106   
## 3+: 51   
##   
##   
## applicant\_income coapplicant\_income loan\_amount loan\_amount\_term  
## Min. : 150 Min. : 0 Min. : 9.0 Min. : 12.0   
## 1st Qu.: 2878 1st Qu.: 0 1st Qu.:100.0 1st Qu.:360.0   
## Median : 3812 Median : 1188 Median :127.0 Median :360.0   
## Mean : 5403 Mean : 1621 Mean :145.3 Mean :342.4   
## 3rd Qu.: 5795 3rd Qu.: 2297 3rd Qu.:165.0 3rd Qu.:360.0   
## Max. :81000 Max. :41667 Max. :700.0 Max. :480.0   
## credit\_history property\_area loan\_status   
## Bad : 93 Rural :179 Approved :422   
## Good:521 Semiurban:233 Not Approved:192   
## Urban :202   
##   
##   
##

* Of all the customers, females were 113 while males were 501, 401 were married while 213 were not married, 480 were graduates while 134 were not graduates, 82 were self employed while 532 were not self employed.
* 354 customers had zero dependents, 103 had 1 dependent, 106 had 2 dependents while 51 had 3 or more dependents. 521 customers had good credit history while 93 had bad credit history. 179 customers had properties in rural settings, 233 had properties in semi-urban settings while 202 had properties in urban settings. 422 loans were approved while 192 weren’t approved.
* The median values for applicant’s income, co-applicant income, loan amount and loan amount term are 3812, 1188, 127 and 360 respectively.

# Visualize the data  
  
# Select the categorical features and convert them to long format  
Cat\_Untidy <- Data |> select(gender, married, dependents,   
 education, self\_employed,  
 credit\_history, property\_area,   
 loan\_status) |>  
 gather(key = "Variable", value = "Value", -loan\_status)

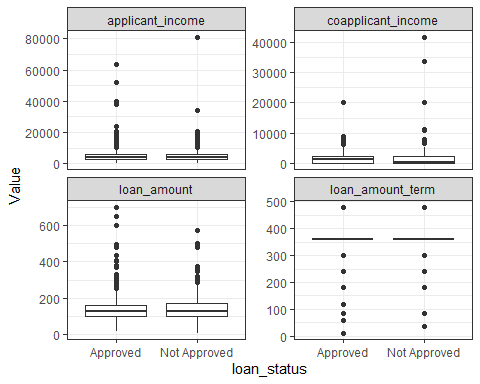
## Warning: attributes are not identical across measure variables; they will be  
## dropped

# Plot  
ggplot(Cat\_Untidy, aes(Value, fill = loan\_status)) +   
 facet\_wrap(~Variable, scales = "free\_x") +  
 geom\_bar(position = "dodge") + theme\_bw()



* Most customers with good credit history had their loan requests approved.
* Most approved loans were for customers who had zero dependents.
* Most of the approved loans were for graduates.
* Most approved loans were for male customers.
* Most approved loans were for married customers.
* Also, most approved loans were for customers who weren’t self-employed.

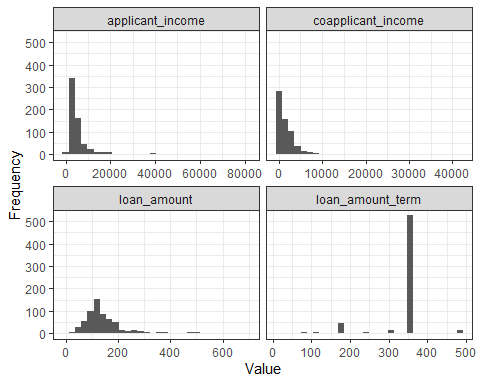
# Select the numeric features plus the response var and convert them to long format  
Num\_Untidy <- Data |> select(applicant\_income, coapplicant\_income,  
 loan\_amount, loan\_amount\_term,  
 loan\_status) |>  
 gather(key = "Variable", value = "Value", -loan\_status)  
  
# Plot  
ggplot(Num\_Untidy, aes(loan\_status, as.numeric(Value))) +   
 facet\_wrap(~Variable, scales = "free\_y") +  
 geom\_boxplot() + theme\_bw() + labs(y = "Value")



The medians for applicant income, co-applicant income and loan amount were generally low, but had some outliers. On average, most of the approved loans were for individuals who needed lower amounts. However, there are individuals who requested for high loan amounts and their loans were approved. Based on the distribution of the numeric features, the data doesn’t seem to be easily separable.

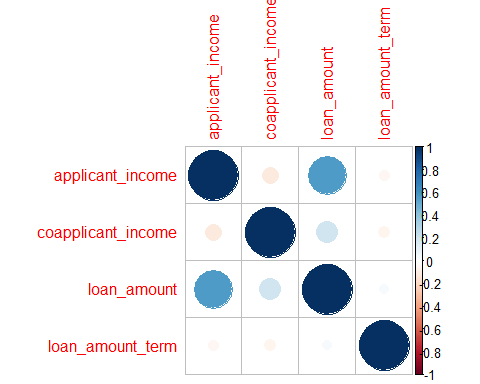
# Plot a histogram of the numeric features  
ggplot(Num\_Untidy, aes(as.numeric(Value))) +   
 facet\_wrap(~Variable, scales = "free\_x") +  
 geom\_histogram() + theme\_bw() +   
 labs(x = "Value", y = "Frequency")

## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.



Applicant income, co-applicant income and loan amount are right skewed while loan amount term is left skewed.

# Check for highly correlated features  
  
# Select numeric features  
Numeric\_features <- Data |> select(applicant\_income,   
 coapplicant\_income,   
 loan\_amount,   
 loan\_amount\_term)  
  
# Generate correlation plot  
corrplot(cor(Numeric\_features))



There is no multicollinearity between the numeric features.

# Feature Engineering

I’ll begin by partitioning the data into training and validation sets using 80/20 split, then prepare the two sets separately to prevent information leakage. I’ll also encode the factor variables to numeric because some algorithms like KNN, SVM and XGBoost cannot handle categorical predictors.

## Partition the data into training and test sets  
  
# Assign data a different name  
Encoded\_data <- Data  
  
# Set seed for reproducibility  
set.seed(42)  
  
# Split the data (use 80/20 split)  
train\_index <- createDataPartition(Encoded\_data$loan\_status, p = 0.80,   
 list = FALSE)  
# Assign 80% to training set  
training\_data <- Encoded\_data[train\_index, ]  
# Assign the remaining 20% to test set  
test\_data <- Encoded\_data[-train\_index, ]

## Prepare training data  
  
# Encode gender  
training\_data[["gender"]] <- factor(training\_data[["gender"]],   
 labels = c(1,2),   
 levels = c("Female", "Male"))  
# Encode married  
training\_data[["married"]] <- factor(training\_data[["married"]],   
 labels = c(0,1),   
 levels = c("No", "Yes"))  
# Encode dependents  
training\_data[["dependents"]] <- factor(training\_data[["dependents"]],   
 labels = c(0,1,2,3),   
 levels = c("0", "1", "2", "3+"))  
# Encode education  
training\_data[["education"]] <- factor(training\_data[["education"]],   
 labels = c(1,0),   
 levels = c("Graduate", "Not Graduate"))  
# Encode self\_employed  
training\_data[["self\_employed"]] <- factor(training\_data[["self\_employed"]],   
 labels = c(0,1),   
 levels = c("No", "Yes"))  
# Encode credit\_history  
training\_data[["credit\_history"]] <- factor(training\_data[["credit\_history"]],   
 labels = c(0,1),   
 levels = c("Bad", "Good"))  
# Encode property\_area  
training\_data[["property\_area"]] <- factor(training\_data[["property\_area"]],   
 labels = c(1,2,3),   
 levels = c("Rural", "Semiurban", "Urban"))

# Convert the encoded predictor variables to numeric  
predictors <- training\_data |> select(-loan\_status) |>   
 mutate\_if(is.factor, ~ as.numeric(.))  
  
# Add column with the target variable  
training\_data <- predictors |> mutate(loan\_status = training\_data$loan\_status)

## Prepare test data  
  
# Encode gender  
test\_data[["gender"]] <- factor(test\_data[["gender"]],   
 labels = c(1,2),   
 levels = c("Female", "Male"))  
# Encode married  
test\_data[["married"]] <- factor(test\_data[["married"]],   
 labels = c(0,1),   
 levels = c("No", "Yes"))  
# Encode dependents  
test\_data[["dependents"]] <- factor(test\_data[["dependents"]],   
 labels = c(0,1,2,3),   
 levels = c("0", "1", "2", "3+"))  
# Encode education  
test\_data[["education"]] <- factor(test\_data[["education"]],   
 labels = c(1,0),   
 levels = c("Graduate", "Not Graduate"))  
# Encode self\_employed  
test\_data[["self\_employed"]] <- factor(test\_data[["self\_employed"]],   
 labels = c(0,1),   
 levels = c("No", "Yes"))  
# Encode credit\_history  
test\_data[["credit\_history"]] <- factor(test\_data[["credit\_history"]],   
 labels = c(0,1),   
 levels = c("Bad", "Good"))  
# Encode property\_area  
test\_data[["property\_area"]] <- factor(test\_data[["property\_area"]],   
 labels = c(1,2,3),   
 levels = c("Rural", "Semiurban", "Urban"))

# Convert the encoded predictor variables to numeric  
predictors <- test\_data |> select(-loan\_status) |>   
 mutate\_if(is.factor, ~ as.numeric(.))  
  
# Add column with the target variable  
test\_data <- predictors |> mutate(loan\_status = test\_data$loan\_status)

# Model Training

I’ll try six different algorithms i.e. Logistic Regression, Naive Bayes, KNN, RF, SVM and XGBoost.

# Define a classification task  
LoanTask <- makeClassifTask(data = training\_data,   
 target = "loan\_status")

# Logistic Regression model

I’ll use Logistic Regression as my basis model. I’ll make use of cross-validation when training my models to assess the generalization ability of the models on new, unseen data. When setting random seed number for reproducibility, I’ll reset the number to use same seed number. This will make sure that the results are directly comparable.

# Define learner  
logReg <- makeLearner("classif.logreg", predict.type = "prob")

# Train the model  
logRegModel <- train(logReg, LoanTask)

# Cross-validate the model training process  
  
# Set seed for reproducibility  
set.seed(1234)  
  
# Define a 6-fold resampling description  
kFold <- makeResampleDesc(method = "RepCV", folds = 6,   
 reps = 60, stratify = TRUE)  
  
# Cross-validate  
logRegCV <- resample(learner = logReg, task = LoanTask,   
 resampling = kFold,   
 measures = list(mmce, acc, fpr, fnr),   
 show.info = FALSE)  
  
# View cross\_validation results  
logRegCV$aggr

## mmce.test.mean acc.test.mean fpr.test.mean fnr.test.mean   
## 0.18806765 0.81193235 0.54749145 0.02425595

The model generalizes well. It has an accuracy of 81.19% and a False Negative Rate of 2.43%. However, FPR is very high.

# Naive Bayes model

# Define learner  
naiveLearner <- makeLearner("classif.naiveBayes", predict.type = "prob")

# Train the model  
bayesModel <- train(naiveLearner, LoanTask)

# Cross-validate the model training procedure  
set.seed(1234)  
bayesCV <- resample(learner = naiveLearner, task = LoanTask,   
 resampling = kFold, measures = list(mmce, acc, fpr, fnr),   
 show.info = FALSE)  
# Check performance  
bayesCV$aggr

## mmce.test.mean acc.test.mean fpr.test.mean fnr.test.mean   
## 0.19355467 0.80644533 0.50919658 0.04970238

The model does not perform better than the Logistic Regression model (has an accuracy of 80.64%). The accuracy is slightly lower than that of Logistic Regression, which is the basis model.

# KNN model

# Define learner  
knnLearner <- makeLearner("classif.knn")

# Define hyperparameter space for tuning k  
knnParamSpace <- makeParamSet(makeDiscreteParam("k", values = 1:30))  
  
# Define search strategy  
gridSearch <- makeTuneControlGrid()  
  
# Define CV for tuning  
cvForTuning <- makeResampleDesc("RepCV", folds = 6, reps = 60, stratify = TRUE)

# Set seed for reproducibility  
set.seed(1234)  
  
# Tune the model with cross-validation  
tunedK <- tuneParams(learner = knnLearner, task = LoanTask,   
 resampling = cvForTuning,   
 par.set = knnParamSpace,   
 control = gridSearch,  
 measures = list(mmce, acc, fpr, fnr),   
 show.info = FALSE)  
  
# Obtain the optimal hyperparameter  
tunedK$x

## $k  
## [1] 29

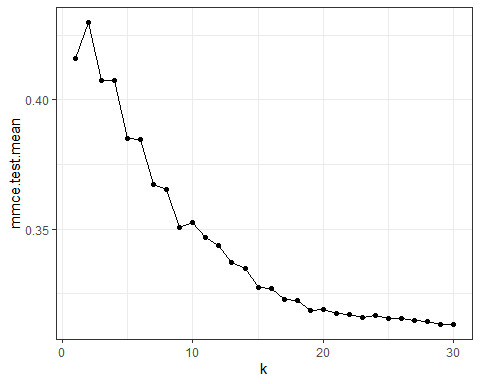
The optimal value of k is 29.

# Print CV results  
tunedK$y

## mmce.test.mean acc.test.mean fpr.test.mean fnr.test.mean   
## 0.313135116 0.686864884 0.997525641 0.001332324

KNN model doesn’t perform well. It has a lower accuracy and a very high false positive rate which is worse. The basis model has a much better performance than this KNN model.

# Extract model information  
knnTuningData <- generateHyperParsEffectData(tunedK)  
  
# Visualize the model tuning process  
plotHyperParsEffect(knnTuningData, x = "k", y = "mmce.test.mean",  
plot.type = "line") +  
theme\_bw()



Mmce value is least at k = 29.

# Set hyperparameters for the final model  
tunedKnn <- setHyperPars(makeLearner("classif.knn"),   
 par.vals = tunedK$x)  
# Train the final model  
tunedKnnModel <- train(tunedKnn, LoanTask)

# Random Forest

# Define learner  
rf\_learner <- makeLearner("classif.randomForest", predict.type = "prob")

# Define hyperparameter space for tuning  
rf\_ParamSpace <- makeParamSet(makeIntegerParam("ntree", lower = 200,   
 upper = 200),  
 makeIntegerParam("mtry", lower = 4,   
 upper = 15),   
 makeIntegerParam("nodesize", lower = 2,   
 upper = 15),  
 makeIntegerParam("maxnodes", lower = 3,   
 upper = 25))

# Define search strategy to use random search with 200 iterations  
randSearch <- makeTuneControlRandom(maxit = 200)  
  
# Define a 6-fold resampling description  
cvForTuning <- makeResampleDesc("CV", iters = 6, stratify = TRUE)

# Begin parallelization  
parallelStartSocket(cpus = detectCores())

## Starting parallelization in mode=socket with cpus=4.

# Set random seed for reproducibility  
set.seed(1234)  
  
# Perform hyperparameter tuning  
tuned\_rf\_Pars <- tuneParams(learner = rf\_learner, task = LoanTask,   
 resampling = cvForTuning,   
 par.set = rf\_ParamSpace,   
 control = randSearch,   
 measures = list(mmce, acc, fpr, fnr),   
 show.info = FALSE)

## Exporting objects to slaves for mode socket: .mlr.slave.options

## Mapping in parallel: mode = socket; level = mlr.tuneParams; cpus = 4; elements = 200.

# Stop parallelization  
parallelStop()

## Stopped parallelization. All cleaned up.

# View cross-validation results  
tuned\_rf\_Pars

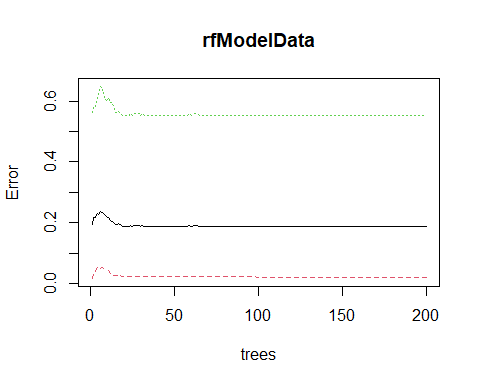
## Tune result:  
## Op. pars: ntree=200; mtry=6; nodesize=7; maxnodes=10  
## mmce.test.mean=0.1869508,acc.test.mean=0.8130492,fpr.test.mean=0.5520513,fnr.test.mean=0.0207289

The random Forest model also generalizes well. It has a mean misclassification error rate of 18.69%, which is slightly lower than that of the Logistic Regression model. The model however, has a higher false positive rate.

The Random Forest model has a slightly better performance than the basis model (Logistic regression model).

# Set the optimal hyperparameters for the final model  
tuned\_rf <- setHyperPars(rf\_learner, par.vals = tuned\_rf\_Pars$x)  
  
# Train the final model using the optimal hyperparameters  
tuned\_rf\_Model <- train(tuned\_rf, LoanTask)

# Check if there are enough trees in the Random Forest  
  
# First extract model information  
rfModelData <- getLearnerModel(tuned\_rf\_Model)  
  
# Plot mmce vs number of trees  
plot(rfModelData)



The mean out-of-bag error stabilizes too early, at about 30 trees. I have enough number of trees in the forest (in fact many). The positive class has a very high mean out-of-bag error rate.

# SVM model

# Define learner  
svmLearner <- makeLearner("classif.svm", predict.type = "prob")

# Define hyperparameter space for tuning the model  
kernels <- c("polynomial", "radial", "sigmoid")  
svmParamSpace <- makeParamSet(makeDiscreteParam("kernel", values = kernels),   
 makeIntegerParam("degree", lower = 1, upper = 4),   
 makeNumericParam("cost", lower = 0.1, upper = 12),   
 makeNumericParam("gamma", lower = 0.1, 7))

# Define search strategy to use random search with 100 iterations  
# Note that SVM is computationally expensive  
randSearch <- makeTuneControlRandom(maxit = 100)  
  
# Define CV strategy  
cvForTuning <- makeResampleDesc("CV", iters = 6, stratify = TRUE)

# Set random seed for reproducibility  
set.seed(1234)  
  
# Start parallelization  
parallelStartSocket(cpus = detectCores())

## Starting parallelization in mode=socket with cpus=4.

# Perform hyperparameter tuning with cross-validation  
tunedSvmPars <- tuneParams(learner = svmLearner, task = LoanTask,   
 resampling = cvForTuning,   
 par.set = svmParamSpace,   
 control = randSearch,  
 measures = list(mmce, acc, fpr, fnr),   
 show.info = FALSE)

## Exporting objects to slaves for mode socket: .mlr.slave.options

## Mapping in parallel: mode = socket; level = mlr.tuneParams; cpus = 4; elements = 100.

# Stop parallelization  
parallelStop()

## Stopped parallelization. All cleaned up.

# View tuning results  
tunedSvmPars

## Tune result:  
## Op. pars: kernel=polynomial; degree=1; cost=8.35; gamma=0.69  
## mmce.test.mean=0.1869508,acc.test.mean=0.8130492,fpr.test.mean=0.5520513,fnr.test.mean=0.0207289

SVM with polynomial kernel of degree 1 is the optimal model. The SVM model has the same performance as Random Forest (mmce value of 18.69%, accuracy of 81.3%).

# Use the optimal hyperparameters to train the final model  
  
# Set the optimal hyperparameters for the final model  
tunedSvm <- setHyperPars(learner = svmLearner, par.vals = tunedSvmPars$x)  
  
# Train the final model  
tunedSvmModel <- train(tunedSvm, LoanTask)

# XGBoost

# Define learner  
XGB <- makeLearner("classif.xgboost", predict.type = "prob")

# Define hyperparameter space for tuning the model  
xgbParamSpace <- makeParamSet(  
makeNumericParam("eta", lower = 0, upper = 1),  
makeNumericParam("gamma", lower = 0, upper = 7),  
makeIntegerParam("max\_depth", lower = 1, upper = 10),  
makeNumericParam("min\_child\_weight", lower = 1, upper = 10),  
makeNumericParam("subsample", lower = 0.5, upper = 1),  
makeNumericParam("colsample\_bytree", lower = 0.5, upper = 1),  
makeIntegerParam("nrounds", lower = 20, upper = 100))

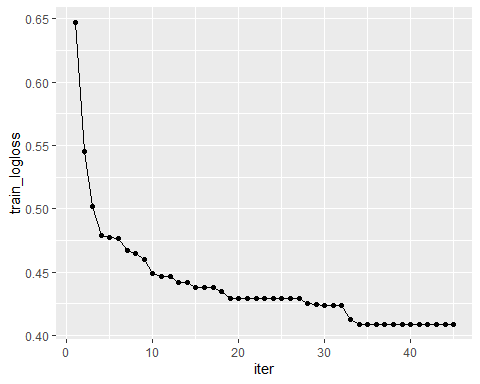
# Define search strategy to use random search  
randSearch <- makeTuneControlRandom(maxit = 700)  
  
# Make resampling description for CV  
cvForTuning <- makeResampleDesc("CV", iters = 6, stratify = TRUE)  
  
# Set random seed for reproducibility  
set.seed(1234)  
  
# Tune the model with cross-validation  
tunedXgbPars <- tuneParams(learner = XGB, task = LoanTask,   
 resampling = cvForTuning,   
 par.set = xgbParamSpace,   
 control = randSearch,  
 measures = list(mmce, acc, fpr, fnr),   
 show.info = FALSE)  
# Check performance  
tunedXgbPars$y

## mmce.test.mean acc.test.mean fpr.test.mean fnr.test.mean   
## 0.18286062 0.81713938 0.53897436 0.02072891

XGBoost performs better than Logistic Regression, SVM and RF. XGBoost outperforms all the other algorithms.

# Train the final model using optimal hyperparameters  
  
# Set the optimal hyperparameters for the final model  
tunedXgb <- setHyperPars(XGB, par.vals = tunedXgbPars$x)  
  
# Train the final model  
tunedXgbModel <- train(tunedXgb, LoanTask)

# Check if there are enough trees for the model  
  
# Extract model information  
xgbModelData <- getLearnerModel(tunedXgbModel)  
  
# Plot  
ggplot(xgbModelData$evaluation\_log, aes(iter, train\_logloss)) +   
 geom\_line() + geom\_point()



Log loss stabilizes after the 34th iteration. I used enough trees.

# Model Validation

I’ll use the best three performing models to make predictions on test data.

# Use the RF model to make predictions on test data  
rfPreds <- predict(tuned\_rf\_Model, newdata = test\_data)  
  
# Collect prediction  
rfPreds\_data <- rfPreds$data

# Calculate confusion matrix  
confusionMatrix(table(rfPreds\_data$truth, rfPreds\_data$response))

## Confusion Matrix and Statistics  
##   
##   
## Approved Not Approved  
## Approved 83 1  
## Not Approved 21 17  
##   
## Accuracy : 0.8197   
## 95% CI : (0.7398, 0.8834)  
## No Information Rate : 0.8525   
## P-Value [Acc > NIR] : 0.8732   
##   
## Kappa : 0.5088   
##   
## Mcnemar's Test P-Value : 5.104e-05   
##   
## Sensitivity : 0.7981   
## Specificity : 0.9444   
## Pos Pred Value : 0.9881   
## Neg Pred Value : 0.4474   
## Prevalence : 0.8525   
## Detection Rate : 0.6803   
## Detection Prevalence : 0.6885   
## Balanced Accuracy : 0.8713   
##   
## 'Positive' Class : Approved   
##

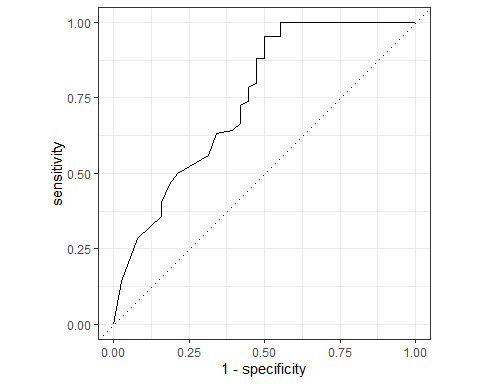
Random Forest model has a validation accuracy of 81.97%. This model is good at identifying customers who do not qualify for a loan (has a Specificity of 94.4%).

# Calculate ROC AUC  
rfPreds\_data |> roc\_auc(truth = truth, prob.Approved)

## # A tibble: 1 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 roc\_auc binary 0.746

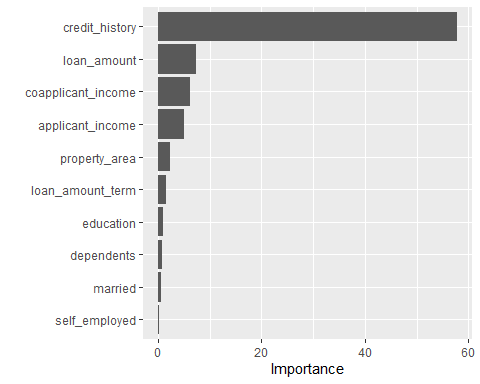
The ROC AUC value for RF isn’t very good.

# Plot ROC curve  
rfPreds\_data |> roc\_curve(truth = truth, prob.Approved) |> autoplot()



The ROC curve looks good, even though it’s further from the top left corner where ROC AUC value is 1.

# Variable importance plot for RF  
vip(tuned\_rf\_Model)



Based on the Random Forest algorithm, the most important predictors of loan approval status include credit history, loan amount, co-applicant income, applicant’s income, property area, loan amount term, education, number of dependents, marital status and self employment respectively.

# Use the SVM model to make predictions on test data  
svmPreds <- predict(tunedSvmModel, newdata = test\_data)  
  
# Collect prediction  
svmPreds\_data <- svmPreds$data

# Calculate confusion matrix  
confusionMatrix(table(svmPreds\_data$truth, svmPreds\_data$response))

## Confusion Matrix and Statistics  
##   
##   
## Approved Not Approved  
## Approved 84 0  
## Not Approved 21 17  
##   
## Accuracy : 0.8279   
## 95% CI : (0.749, 0.8902)  
## No Information Rate : 0.8607   
## P-Value [Acc > NIR] : 0.8784   
##   
## Kappa : 0.5271   
##   
## Mcnemar's Test P-Value : 1.275e-05   
##   
## Sensitivity : 0.8000   
## Specificity : 1.0000   
## Pos Pred Value : 1.0000   
## Neg Pred Value : 0.4474   
## Prevalence : 0.8607   
## Detection Rate : 0.6885   
## Detection Prevalence : 0.6885   
## Balanced Accuracy : 0.9000   
##   
## 'Positive' Class : Approved   
##

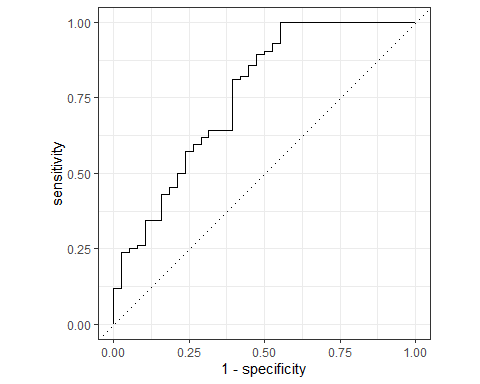
SVM model has a validation accuracy of 82.79%. This model is very good at identifying customers who do not qualify for a loan (has a Specificity of 1). Precision(PPV) of 1 is very good, and a Sensitivity of 0.8 is also good.

# Calculate ROC AUC  
svmPreds\_data |> roc\_auc(truth = truth, prob.Approved)

## # A tibble: 1 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 roc\_auc binary 0.760

ROC AUC value of 0.759 isn’t that bad.

# Plot ROC curve  
svmPreds\_data |> roc\_curve(truth = truth, prob.Approved) |> autoplot()



# Use the XGB model to make predictions on test data  
xgbPreds <- predict(tunedXgbModel, newdata = test\_data)  
  
# Collect prediction  
xgbPreds\_data <- xgbPreds$data

# Calculate confusion matrix  
confusionMatrix(table(xgbPreds\_data$truth, xgbPreds\_data$response))

## Confusion Matrix and Statistics  
##   
##   
## Approved Not Approved  
## Approved 83 1  
## Not Approved 20 18  
##   
## Accuracy : 0.8279   
## 95% CI : (0.749, 0.8902)  
## No Information Rate : 0.8443   
## P-Value [Acc > NIR] : 0.7394   
##   
## Kappa : 0.535   
##   
## Mcnemar's Test P-Value : 8.568e-05   
##   
## Sensitivity : 0.8058   
## Specificity : 0.9474   
## Pos Pred Value : 0.9881   
## Neg Pred Value : 0.4737   
## Prevalence : 0.8443   
## Detection Rate : 0.6803   
## Detection Prevalence : 0.6885   
## Balanced Accuracy : 0.8766   
##   
## 'Positive' Class : Approved   
##

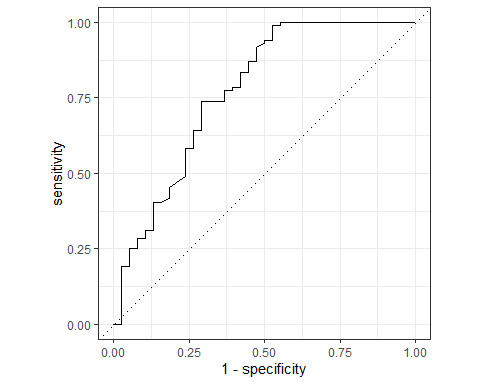
XGBoost has the same validation accuracy as the SVM model. XGBoost has a slightly better Sensitivity than SVM, but a slightly lower Precision and Specificity.

# Calculate ROC AUC  
xgbPreds\_data |> roc\_auc(truth = truth, prob.Approved)

## # A tibble: 1 × 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 roc\_auc binary 0.772

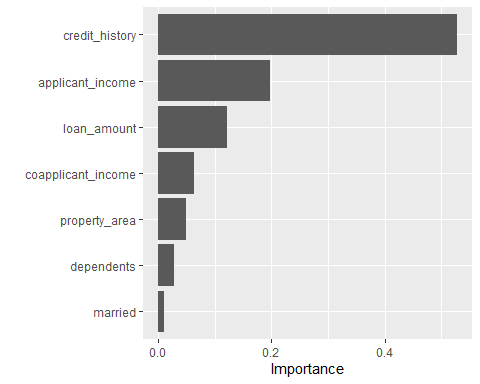
XGBoost model has a better ROC AUC value than SVM model (0.77 compared to 0.76).

# Plot ROC curve  
xgbPreds\_data |> roc\_curve(truth = truth, prob.Approved) |> autoplot()



The lower part of the ROC curve doesn’t look good.

# Variable importance plot for XGBoost  
vip(tunedXgbModel)



The most important predictors of loan approval status are credit history, applicant income, loan amount, co-applicant income, property area, number of dependents and marital status respectively.

* I’ll pick the SVM model because it has the best Precision and Specificity. The model is good at identifying/detecting customers who do not qualify for loans. This is in-line with my objective, which was to build a classification model that can help banks and other financial institutions to mitigate loses incurred from bad loans.
* NB: The limitation of this analysis however, is that I did not handle class imbalance in the data.

# Model Application

The model will be useful for banks and money lending companies, as it will help them to identify customers who do not qualify for loans. This will prevent them from issuing bad loans hence reducing losses incurred from loan defaults.