Machine Learning Overview with Lending Club Dataset

Enzo Profili

2019-10-26

Contents

1 Introduction

In Part II, we will dig deeper in some important concepts that can make the algorithm more accurate and customizable. We will now use the Kaggle Lending Club dataset, which is much larger - more than 800,000 rows - and consequently computer speed will start to become a factor, so keeping the code as fast as possible is crucial. Of course, there are services such as Google Graphics Processing Unit and Amazon High Performance Computing that lend processing power.

2 Data Cleaning

2.1 Read in the data

Using the fread function (from the package data.table) is generally good practice, since it is generally faster than the read.csv function from base R.

```
library(tidyverse)
library(data.table)
loans <- fread('~/tex/dataexploration/data/loan.csv',
               sep = ",",header = TRUE)
```

2.2 Target Variable

The first step in any sucessful machine learning is understanding the data and adapting it for the model - this is called data cleaning. In this dataset, our objective is to predict whether a loan will be paid in time. So, first of all, let's understand our target variable - loan status. There are 10 categories, as shown in Table 1.

Table 1: Loan status categories in the Lending Club data.

Category	Frequency
Charged Off	45248
Current	601779
Default	1219
Does not meet the credit policy. Status:Charged Off	761
Does not meet the credit policy. Status:Fully Paid	1988
Fully Paid	207723
In Grace Period	6253
Issued	8460
Late (16-30 days)	2357
Late (31-120 days)	11591

To make it simpler, let's split the dataframe into good and bad loans; bad loans include defaulted and charged off, while good ones were fully paid; the other class (such as current loans or late for a short period of time) were ignored, since they do not provide enough insight on whether a loan is good or bad, and might reduce the signal from actually defaulted loans.

```
# filter ambiguous status types

loans <- filter(loans, loan_status != "Issued",
                loan_status != "Current",
                loan_status != "Does not meet the credit policy. Status:Charged Off",
                loan_status != "Does not meet the credit policy. Status:Fully Paid",
                loan_status != "In Grace Period",
                loan_status != "Late (16-30 days)",
                loan_status != "Late (31-120 days)")

# divide loan types into good and bad
loans <- mutate(loans, loan_quality = ifelse(loan_status == "Fully Paid",
                                             "Good", "Bad"))
loans$loan_quality <- as.factor(loans$loan_quality)
```

2.3 Predictor Variables

After, we need to check for the predictor variables. We have to:

- delete data that is impractical to work with (such as ones with long explanations by the borrower)
- delete data that spills information from the future (since the model will not have future information to predict loans)
- delete variables with too many missing values
- delete data with insufficient variation (only one value, for example)
- see if there are variables that require manipulation or filtering to meet our needs

Some new variables were created out of the initial ones. For example, the variable year, which states the year the loan was taken (and might be useful since macroeconomic conditions vary), and the amount of years between the loan was taken and the earliest credit line for the client was created (older accounts may be less risky). Some new variables were taken from Polena (2017).[1] The list of variables that remained are in the remained variables definition section, Section 13. This section also contains the description of all created variables.

[1] See Polena (2017) chapter 5 for more details.

```
# new variable: income to amount relationship
loans <- mutate(loans,
                loan_income_ratio = loans$annual_inc/loans$loan_amnt)

# new variable: collection in last 12 months
loans <- mutate(loans,
                collections = ifelse(collections_12_mths_ex_med > 0, 1, 0))
loans$collections <- as.factor(loans$collections)

# new variable: last credit pulled
loans <- mutate(loans,
                last_credit_pull_d_v2 = str_sub(last_credit_pull_d,-4,-1))
loans$last_credit_pull_d_v2 <- as.factor(loans$last_credit_pull_d_v2)

# new variable: loan year
loans <- mutate(loans, year = str_sub(issue_d,-4,-1))
loans$year <- as.numeric(loans$year)

# new variable: credit line age
loans <- mutate(loans,
                credit_line_age = as.numeric(year)
                - as.numeric(str_sub(earliest_cr_line,-4,-1)))
loans$year <- as.factor(loans$year
                        )
# new variable: description length
loans <- mutate(loans, desc_length = nchar(desc))
loans$desc_length <- as.numeric(loans$desc_length)
```

Another variable of potential interest is `desc`. This field contains explanations about the borrowing provided by the borrower. Obviously this will not be standardized and it will be difficult to interpret, but it might be interesting to see if the length of the explanation has any explanatory power. Thus, we create `desc_length`, which is the number of characters in the field.

Finally, we remained with 26 variables out of the initial 74. Many variables were removed because they report future characteristics of the loan, or are unique to each loan. Descriptions of all removed variables are in the removed variables definition section, Section 14.

```
# filter irrelevant/future features
drop_cols <- c('desc','url', 'id', 'member_id', 'funded_amnt',
               'funded_amnt_inv', 'grade', 'int_rate', 'emp_title',
               'zip_code', 'out_prncp', 'out_prncp_inv', 'total_pymnt',
```

```
                'total_pymnt_inv', 'total_rec_prncp','total_rec_int',
                'total_rec_late_fee', 'recoveries',
                'collection_recovery_fee', 'last_pymnt_d',
                'last_pymnt_amnt', 'title', 'verification_status_joint',
                'next_pymnt_d', 'issue_d', 'loan_status',
                'last_credit_pull_d', 'earliest_cr_line', 'pymnt_plan',
                'application_type', 'policy code',
                'collections_12_mths_ex_med', 'policy_code',
                'tot_coll_amt', 'collections_12_mths_ex_med',
                'collections')
loans <- loans %>% dplyr::select(-one_of(drop_cols))
```

2.4 Missing Values

Delete variables that are predominantly missing; see Table 2.

```
library(purrr)
missing <- map_dbl(loans, ~sum(is.na(.))/length(.))
loans <- loans[missing < 0.5]
kable(missing[missing > 0.5], caption = "Variables that have been deleted for numerou
    col.names = c("Fraction missing"), label = "missing",
    longtable = TRUE, booktabs = TRUE) %>%
    kable_styling(latex_options = c("hold_position"))
```

Table 2: Variables that have been deleted for numerous missing values

	Fraction missing
mths_since_last_delinq	0.5564578
mths_since_last_record	0.8748417
mths_since_last_major_derog	0.8116527
annual_inc_joint	0.9999961
dti_joint	0.9999961
open_acc_6m	0.9994335
open_il_6m	0.9994335
open_il_12m	0.9994335
open_il_24m	0.9994335
mths_since_rcnt_il	0.9994492
total_bal_il	0.9994335
il_util	0.9995043
open_rv_12m	0.9994335

open_rv_24m	0.9994335
max_bal_bc	0.9994335
all_util	0.9994335
inq_fi	0.9994335
total_cu_tl	0.9994335
inq_last_12m	0.9994335

2.5 Insufficient Variation

We can use the Herfindahl index to measure the variation contained in a variable. It is unclear what to use as a cutoff. Setting it equal to 0.95 eliminates one variable, acc_now_delinq; see Table 3.

```
cutoff_herf <- 0.95
herfindahl <- map_dbl(loans, ~sum(prop.table(table(.))^2))
loans <- loans[herfindahl < cutoff_herf]
kable(herfindahl[herfindahl >= cutoff_herf],
    booktabs = TRUE, col.names = "Herfindahl index",
    caption = paste0("Herfindahl indices exceeding ",
        cutoff_herf), label = "herf") %>%
    kable_styling(latex_options = c("hold_position"))
```

Table 3: Herfindahl indices exceeding 0.95

	Herfindahl index
acc_now_delinq	0.9941858

```
# remove factor levels with one
# observation (to avoid errors in the
# logistic regression down the road)
loans <- loans[loans$addr_state != "IA",
    ]
```

2.6 Collinearity/Scaling

Another assesment that should be made before proceeding is predictor collinearity. If two variables are highly correlated, then having both is redundant and will only make the model more unstable and computationally expensive. Therefore, it is advised to remove one of the correlated variables. Let's check the correlation matrix for our predictors:

```
corrplot(cor(loans[complete.cases(loans), sapply(loans, is.numeric)]))
```

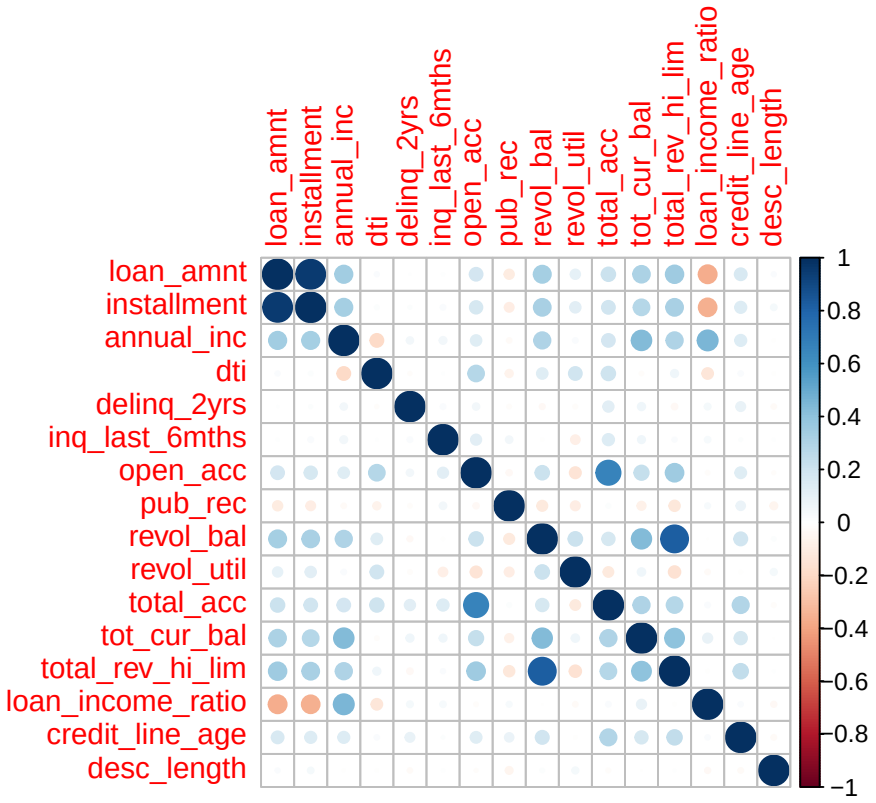


Figure 1: Predictor correlation plot

```
loans <- loans %>% dplyr::select(-loan_amnt)
```

Notice there is very large correlation between the predictors loan_amnt and installment, so let's remove loan_amnt from the dataset (as it is slightly more correlated with other predictors). Apart from that, we are good to go.

One last point to be made is about data scaling. Since we have numeric predictors with many different values (e.g. income ranges from 0 to 300,000 while age ranges from 0 to 50), the ones with larger values (such as income) may influence more the final result. This happens especially in neural networks, but not in random forest (or logistic regression), as it only needs the absolute value to split nodes (little data preparation is an advantage of random forests). One drawback of normalization is that regression coefficients no longer make sense. The default for the scale function in R is the following:

$$x' = \frac{x - \bar{x}}{\sigma}$$

σ represents the column standard deviation, and $\bar{x}$ is the column mean. We will be running neural networks in this report, so I have scaled the data with the scale function, following the procedure in Beque and Lessman (2017). We will compare results between normalized and unnormalized data for random forests and neural networks to show the impact of normalization.[2]

```
# we need to add function(x) description so that scale does not return
# attributes, and type remains unchanged
scale.loans <- loans %>% mutate_if(is.numeric,
                                   function(x) as.numeric(scale(x)))
```

2.7 The end result

This leaves us with 26 variables and 254184 observations; see Table 4:

Table 4: Variables in the final data set

Variable	Missing
term	0
installment	0
sub_grade	0
emp_length	0
home_ownership	0
annual_inc	0
verification_status	0
purpose	0
addr_state	0
dti	0
delinq_2yrs	0
inq_last_6mths	0
open_acc	0
pub_rec	0
revol_bal	0
revol_util	200
total_acc	0
initial_list_status	0
tot_cur_bal	63721
total_rev_hi_lim	63721
loan_quality	0

[2]For more data pre-processing techniques analysis, I recommend checking Crone and Stahlbock (2005).

loan_income_ratio	0
last_credit_pull_d_v2	0
year	0
credit_line_age	0
desc_length	0

3 Data Analysis

Now that we have adjusted our data, let's analyze it. We find many factors that may be important to predict bad loans, such as credit score, loan purpose, location and employment stability. So let's see if that's the case!

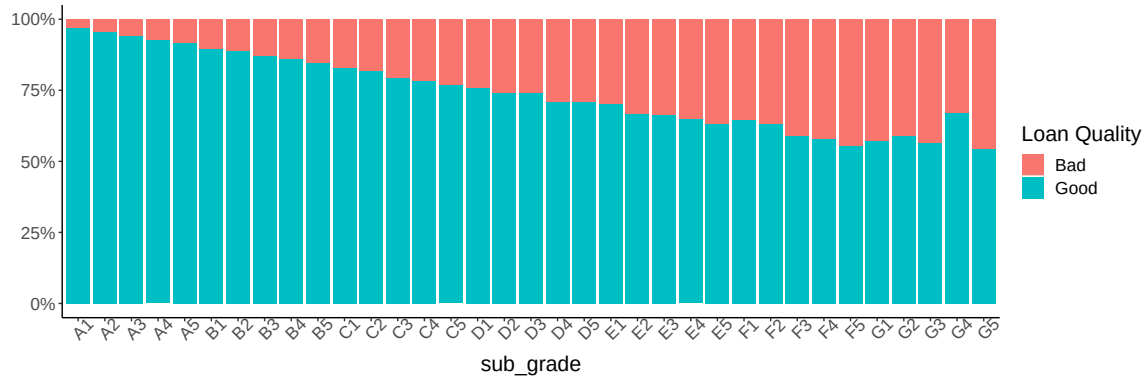


Figure 2: Default rates for credit subgrades

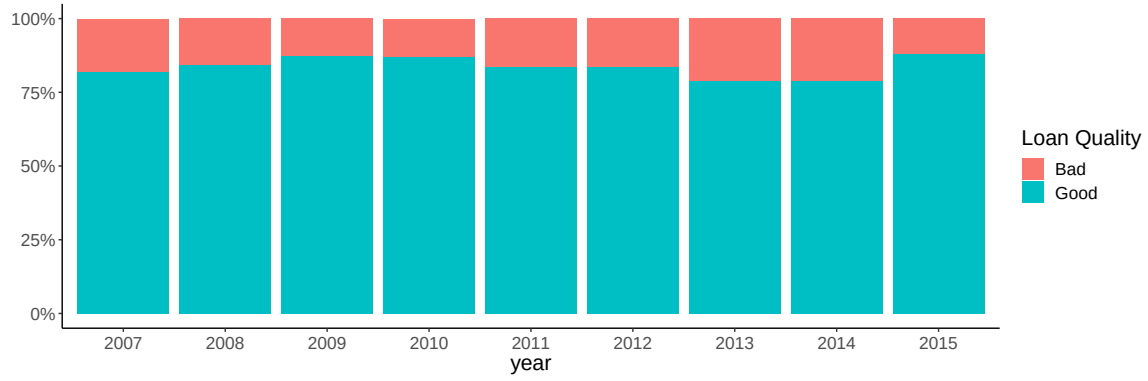


Figure 3: Default rates by year loan started

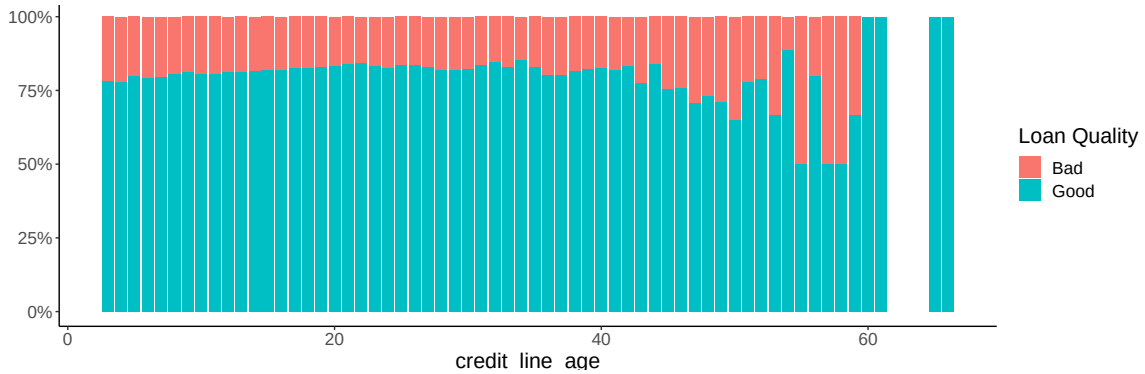


Figure 4: Default rates by credit line age

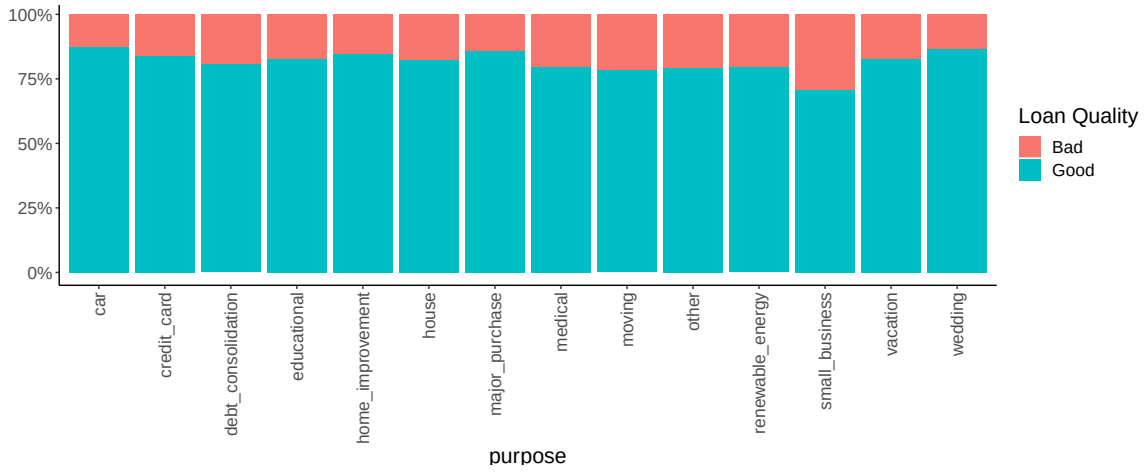


Figure 5: Default rates by purpose

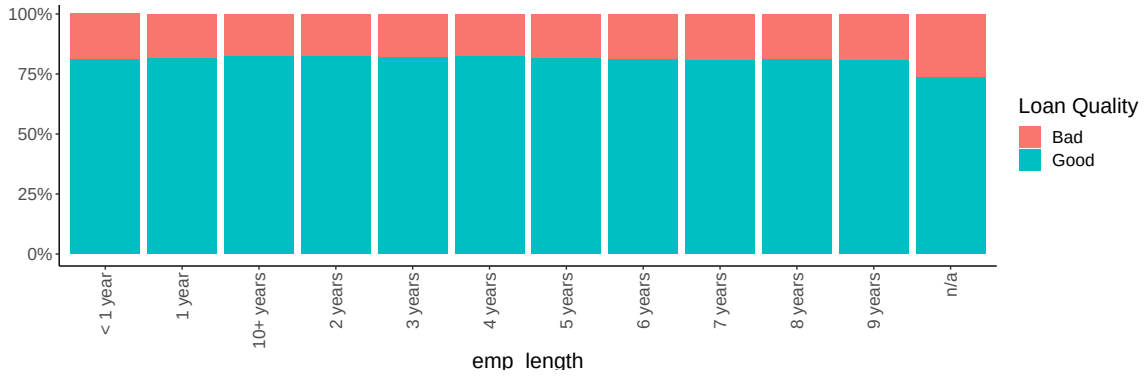


Figure 6: Default rates by employment length

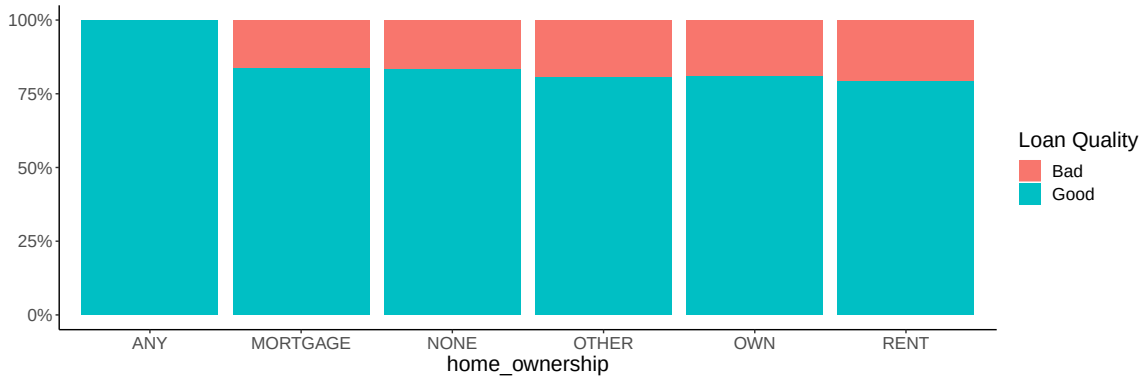


Figure 7: Default rates by home ownership

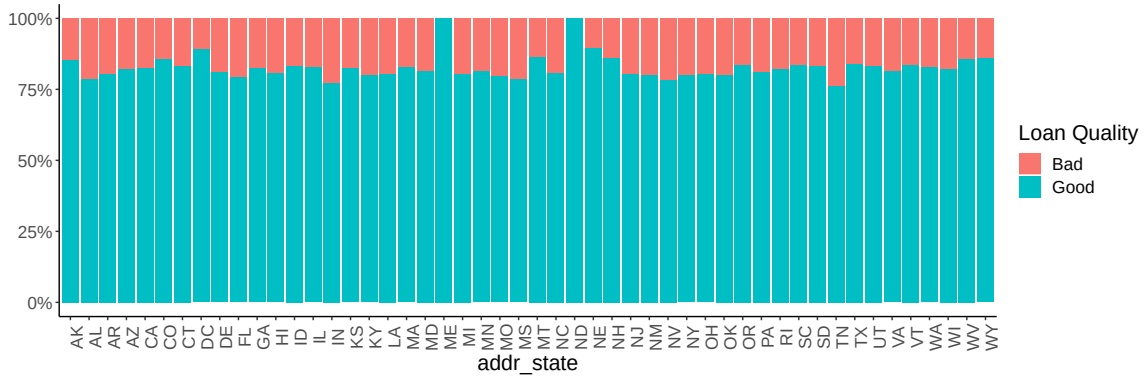


Figure 8: Default rates by state

From these graphs, we see that credit grade and credit line age show a meaningful relationship with default rates. Not surprisingly, higher credit grades determine less bad loans. The other variables might possibly be helpful, but the pattern is less clear. Now, let's analyze continuous variables, such as income, with density plots:

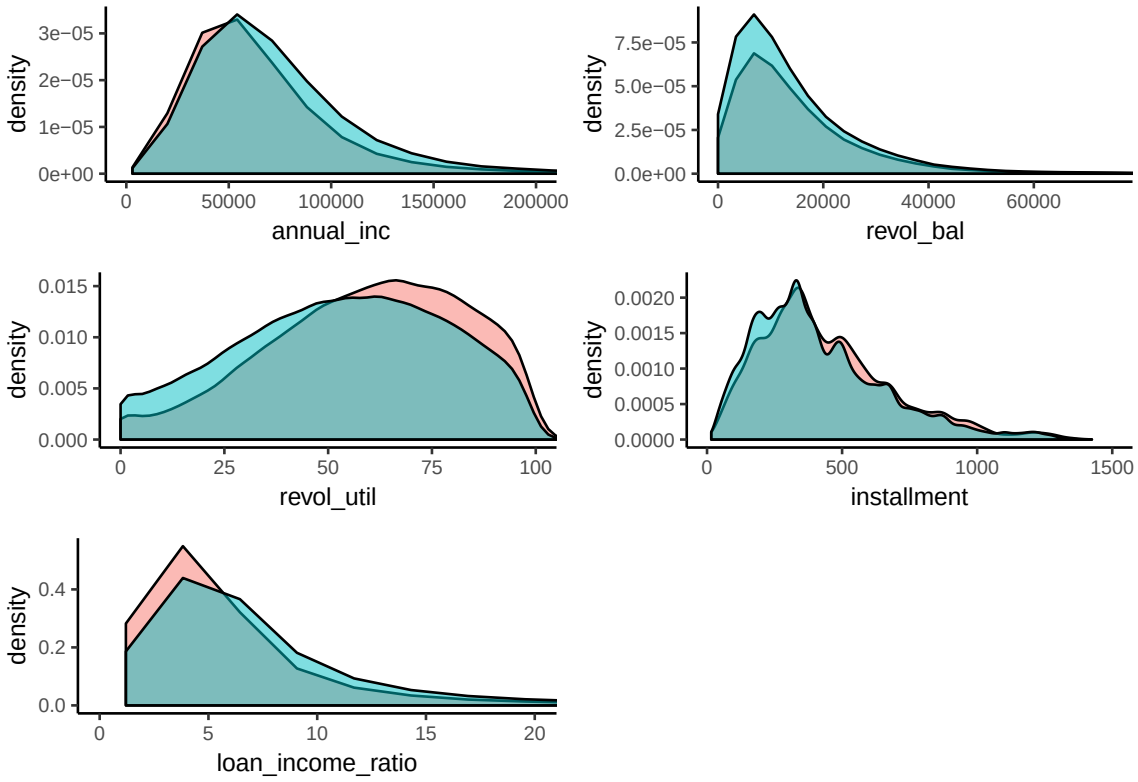


Figure 9: Density plots of default rates against various continuous predictors

These plots show us some interesting data features. We see that income and loan-income ratio show considerable difference between good and bad loans. The percentage of revolving credit used and loan amount too. So, these variables might be relevant to address defaults. These are good indicators that the data is somewhat useful to explain defaulting.

4 Data Partitioning

So, let's generate models with all variables deemed relevant. Only a subset of the data was used, since it makes calculations faster and does not interfere significantly on the model's accuracy. We then split the data between training and testing datasets.

```
# take only 200000 rows (saves processing power)
set.seed(1981)
indexes <- sample(nrow(loans), 200000)
loans_subset <- loans[indexes,]
loans_subset <- as.data.frame(unclass(loans_subset))
```

```
scale.loans_subset <- scale.loans[indexes,]
scale.loans_subset <- as.data.frame(unclass(scale.loans_subset))

# split the data into two similar sets, one for training and one for
# testing (explained later), keeping similar default ratio
set.seed(2437)
splitIndex <- createDataPartition(loans_subset$loan_quality, p = .70,
                                  list = FALSE,
                                  times = 1)
trainSplit <- loans_subset[ splitIndex,]
testSplit <- loans_subset[-splitIndex,]
scale.trainSplit <- scale.loans_subset[ splitIndex,]
scale.testSplit <- scale.loans_subset[-splitIndex,]

# filter for complete cases in both training and test datasets
trainSplit <- trainSplit %>% filter(complete.cases(trainSplit))
testSplit <- testSplit %>% filter(complete.cases(testSplit))

scale.trainSplit <- scale.trainSplit %>% filter(complete.cases(scale.trainSplit))
scale.testSplit <- scale.testSplit %>%
filter(complete.cases(scale.testSplit))

# create matrix of predictors
rf_predictors <- trainSplit[, !names(trainSplit)
                               %in% c("loan_quality")]
rf_predictors_test <- testSplit[, !names(testSplit)
                                  %in% c("loan_quality")]

scale.rf_predictors <- scale.trainSplit[,!names(scale.trainSplit)
                                        %in% c("loan_quality")]
scale.rf_predictors_test <- scale.testSplit[, !names(scale.testSplit)
                                            %in% c("loan_quality")]
```

When reducing the dataset, you will notice the data was split into two datasets: trainSplit and testSplit. The names are self-explanatory, but trainSplit is used to train the model, while testSplit is a dataset the model has never seen, and so the model can be accurately tested. This is called the validation set approach. The datasets share a constant bad loan proportion to ensure reproducibility (this is the default setting in caret's createDataPartition function).

Random forests already conduct this testing process, as each tree uses only around two thirds of the data, and the other third is used for testing. The OOB error rate is

an useful estimate of the test error rate, but, whenever possible, it is good to use the test dataset as well.[3]

5 Logistic Regression

As we are trying to predict a binary variable, logistic regression is a viable option to estimate the probability of default. Since we are discussing probabilities, the results must be between 0 and 1. However, note that a linear regression in this case could yield probabilities above 1 or below 0 for some predictor values.[4] So, logit adapts the linear regression by using the exponential of the predictors. Check the following equation:

$$Pr(Y_i = 1|X_i) = \frac{e^{X_i\beta}}{1 + e^{X_i\beta}}$$

It becomes clear that, as $X_i\beta \to \infty$, $Pr(Y_i = 1|X_i) \to 1$, and as $X_i\beta \to -\infty$, $Pr(Y_i = 1|X_i) \to 0$. So, $Pr(Y_i = 1|X_i) \in [0,1]$. In our bank loans example, each $Y_i = 1$ represents a loan that can be assigned either as "Good" or "Bad". The logit model will compute the probability the loan is bad and plot the logit transformation.[5]

```
#run logistic regression
logistic = glm(loan_quality ~ ., data = trainSplit, family = binomial(logit))

# predict loan quality for test dataset and generate confusion matrix
glm.probs = predict(logistic, testSplit, type = "response")
glm.pred = rep("Bad", nrow(testSplit))
glm.pred[glm.probs >.5] = "Good"
kable(table(glm.pred , testSplit$loan_quality))
```

	Bad	Good
Bad	610	505
Good	7943	35818

```
mean(glm.pred == testSplit$loan_quality)
```

```
[1] 0.8117479
```

The model generates a dummy variable for every factor variable (except one for each predictor to avoid multicollinearity), and consequently we have hundreds of predictors. A large portion of them are in fact significant at the 0.1% level, which shows the

[3] check James and Tibshiran (2013) pages 317-319 for more details and an example.

[4] James and Tibshiran (2013) displays a graphical example of this issue in page 131.

[5] See James and Tibshiran (2013) pages 130-137 for a more detailed explanation on logistic regressions.

quality of our manipulated data. We see that the logistic regression has an accuracy rate of 81% on the test dataset. Let's now analyze nonparametric models on this dataset and compare.

6 Random Forests

6.1 Regular Random Forest

Let's run our random forest. The default value for mtry is $\sqrt{p}$, where p is the number of predictors. Since there are 25 predictors in the dataset, mtry = 5. Of course, we could change this value, but let's leave it for now.

```
# run random forest
set.seed(1234)
rf1 <- randomForest(x = rf_predictors,
                    y = trainSplit$loan_quality,
                    importance = TRUE,
                    ntree = 300)
rf1
```

```
Call:
 randomForest(x = rf_predictors, y = trainSplit$loan_quality,      ntree = 300, impor
               Type of random forest: classification
                     Number of trees: 300
No. of variables tried at each split: 5

        OOB estimate of  error rate: 18.98%
Confusion matrix:
     Bad  Good class.error
Bad  608 19533 0.969812820
Good 367 84342 0.004332479
```

```
# run rf with scaled predictors
set.seed(1234)
scale.rf1 <- randomForest(x = scale.rf_predictors,
                    y = scale.trainSplit$loan_quality,
                    importance = TRUE,
                    ntree = 300)
scale.rf1
```

```
Call:
 randomForest(x = scale.rf_predictors, y = scale.trainSplit$loan_quality,      ntree
```

```
               Type of random forest: classification
                     Number of trees: 300
No. of variables tried at each split: 5

        OOB estimate of  error rate: 18.98%
Confusion matrix:
     Bad  Good class.error
Bad  638 19503 0.968323321
Good 402 84307 0.004745659
```

Notice there was almost no difference between the random forests with scaled and non-scaled data, with a 81% OOB accuracy (similar to the logistic regression).

6.2 SMOTE

The random forest test had a relatively small OOB error (19%), but, since bad loans are less frequent (25%), it almost never predicted it, and so the model adds almost no value to the analysis. A possible solution to this problem is over-sampling bad loans and under-sampling good loans in the dataset- this method is called SMOTE (SMOTE: Synthetic Minority Over-sampling Technique). This way, we will have more bad loans and it will be easier to predict them.

In R, the function SMOTE (Synthetic Minority Over-sampling Technique) from the package DMwR does the trick.[6] With this function, you set how much the rare event is oversampled (through the perc.over argument) and the common event is undersampled (through the perc.under argument). In Provost and Weiss (2003), it was shown that the optimal class distribution should contain between 50% and 90% minority class examples within the training set. So, let's stick to this measure (50%, more specifically). Note as well you should only apply SMOTE to the training data, as otherwise you will be generating samples that are closely related to the training set and you will get an overly optimistic accuracy rate.

Notice that, the larger the proportion minority class samples in the training set, the more it will be predicted in the testing dataset. Thus, if classifying a bad loan as good as more costly than the opposite, you should have a training set with many defaulted loans (relative to good ones). See below a random forest applied to a balanced dataset:

```
# balance data with SMOTE
kable(table(trainSplit$loan_quality))
```

Var1	Freq
Bad	20141
Good	84709

[6] For the interested reader, Chawla and Kegelmeyer (2002) explains SMOTE more thoroughly.

```
balanced.data <- SMOTE(loan_quality ~., trainSplit,
                       perc.over = 200, perc.under = 150)
kable(table(balanced.data$loan_quality))
```

Var1	Freq
Bad	60423
Good	60423

```
rf_predictors_balanced <- balanced.data[, !names(balanced.data)
                                        %in% c("loan_quality")]
```

```
# run random forest and plot results
set.seed(1234)
rf2 <- randomForest(x = rf_predictors_balanced,
                    y = balanced.data$loan_quality,
                    importance = TRUE,
                    ntree = 300)
rf2
```

```
Call:
 randomForest(x = rf_predictors_balanced, y = balanced.data$loan_quality,      ntree
               Type of random forest: classification
                     Number of trees: 300
No. of variables tried at each split: 5

        OOB estimate of  error rate: 16.21%
Confusion matrix:
       Bad  Good class.error
Bad  46196 14227  0.23545670
Good  5365 55058  0.08879069
```

Through this technique, we could increase accuracy in predicting bad loans from 2% to 77%, and the OOB error rate actually decreased (though results may vary). Finally, just an explanation on the math in perc.over and perc.under (since it can be confusing). Originally, there were approximately 20000 bad loans. The number of bad loans added is Perc.over/100. So perc.over = 200% triples the original, and so we have 60000 examples. Perc.under adds 200% of new bad loans in perc.over, so that the amount of good loans are twice the amount of bad loans added. So, the number of good loans are 200*200/100 = 400% of the original number of bad loan cases (80000).

6.3 Cutoffs

Another way to predict more bad loans runs through an important concept: cutoffs. They are the proportion at which a vote is decided. By default, random forests have a cutoff of .5, which means that if more than 50% of trees decide a loan is bad, it returns "Bad Loan". However, if in your model you would like to catch more bad loans (at the cost of cathing additional good loans unintentionally), you can set a lower cutoff (say, 0 0.2). This way, if 20% or more trees vote for "Bad Loan", the model returns "Bad Loan", even if 80% of the trees voted against it. Below is a random forest with cutoff set to .3:

```
set.seed(1234)
rf3 <- randomForest(x = rf_predictors, y = trainSplit$loan_quality,
    importance = TRUE, cutoff = c(0.2, 0.8),
    ntree = 300)
rf3
```

```
Call:
 randomForest(x = rf_predictors, y = trainSplit$loan_quality,      ntree = 300, cutof
               Type of random forest: classification
                     Number of trees: 300
No. of variables tried at each split: 5

        OOB estimate of  error rate: 31.07%
Confusion matrix:
       Bad  Good class.error
Bad  12731  7410   0.3679063
Good 25164 59545   0.2970641
```

It can be seen above that, by applying a .2 cutoff, more bad loans were predicted (63%), but 30% of good loans were also deemed bad, culminating in a higher OOB error rate (31%). Since most bad loans were still not caught, a stricter cutoff could be implemented. So, this can be a very important tool if used carefully, and can be used to tailor the random forest algorithm to a certain business' needs.

7 ROC Curve

7.1 Explanation

But what if we knew the proportions of bad loans caught for every cutoff level? The ROC (Receiver Operating Characteristic) curve is the answer - a way to assess the performance of the model. Given the first random forest trained (rf1), we will try to predict bad loans with new values (from the dataset testSplit), and see how well the

model performs with these curves. First, some concepts need to be clarified:

a) Sensitivity/recall: the ratio of true positives (positives that the model identified) and the total number of positives (true positives + false negatives). Basically, the ratio of positive results that were correctly predicted. From here on, let's consider this a test for good loans, which means good loans should return "positive" and bad loans should return "negative".

b) Specificity: the ratio of true negatives (negatives that the model identified) and the total number of negatives (true negatives + false positives). Basically, the ratio of negative results that were correctly predicted.

The table below shows what is in each category in a confusion matrix. The row values are the actual ones (so that actual negatives is TN + FP), and the column show predicted values:

	0	1
0	TN	FP
1	FN	TP

Clearly, there is a tradeoff between sensitivity and specificity. In our example, we can simply assume that every loan is good (supposing a "positive" value is a good loan), and thus have zero specificity and one sensivity. In the other hand, we can assume every loan is bad, and have one specificity and zero sensivity. Probably, the ideal result is between one of these two scenarios, and that is what the ROC curve shows.

As can be seen in Figure 10, the y axis is the sensivity and the x axis is 1 - specificity. There is also a line between (0,1) and (1,0), showing the performance of a model that picks results randomly. The triangle below this line has an area under the curve (AUC) of 0.5. A machine learning model's AUC should be as close to 1 as possible, achieving sensivity close to 1 without giving up specificity. Graphically, this would mean the curve "hugs" the top left corner.

7.2 Random Forest Comparison

Given this explanation, let's see how rf1, rf2 and rf3 fared. Since we want to focus on the number of bad loans caught, we have to focus on the specificity measure.

```
rf1.pred = predict(rf1, testSplit, type = "class")
rf1_table = table(testSplit$loan_quality,rf1.pred)
mean1 = mean(rf1.pred == testSplit$loan_quality)
rf2.pred = predict(rf2, testSplit, type = "class")
rf2_table = table(testSplit$loan_quality,rf2.pred)
mean2 = mean(rf2.pred == testSplit$loan_quality)
```

```
rf3.pred = predict(rf3, testSplit, type = "class")
rf3_table = table(testSplit$loan_quality,rf3.pred)
mean3 = mean(rf3.pred == testSplit$loan_quality)
oobs <- c((rf1$confusion[1,1]+rf1$confusion[2,2])/nrow(trainSplit),
          (rf2$confusion[1,1]+rf2$confusion[2,2])/nrow(balanced.data),
          (rf3$confusion[1,1]+rf3$confusion[2,2])/nrow(trainSplit))
rfs <- c("rf1","rf2","rf3")
means <- c(mean1,mean2,mean3)
spec <- c(rf1_table[1,1]/(rf1_table[1,1]+rf1_table[1,2]),
          rf2_table[1,1]/(rf2_table[1,1]+rf2_table[1,2]),
          rf3_table[1,1]/(rf3_table[1,1]+rf3_table[1,2]))
sens <- c(rf1_table[2,2]/(rf1_table[2,1]+rf1_table[2,2]),
          rf2_table[2,2]/(rf2_table[2,1]+rf2_table[2,2]),
          rf3_table[2,2]/(rf3_table[2,1]+rf3_table[2,2]))
rf_results = data.frame(rfs,oobs,means,sens,spec)
  colnames(rf_results) = c("RF", "OOB", "Testing Accuracy",
                           "Sensivity", "Specificity")
kable(rf_results) %>% row_spec(0,bold=TRUE)
```

RF	OOB	Testing Accuracy	Sensivity	Specificity
rf1	0.8102051	0.8117034	0.9972745	0.0236174
rf2	0.8378763	0.7490641	0.8229772	0.4351689
rf3	0.6893276	0.6925751	0.7053933	0.6381387

7.3 Random Forest ROC

Let's see how our initial model (rf1) fares in this analysis. Below, we see that it has an AUC of .73, an interesting start. Bear in mind this is not a direct accuracy measure. The accuracy of the model on the testing dataset was 81%, not 73%. The AUC for logistic regression we ran was .72, slightly lower than the random forest. The AUC for rf2 (the model with SMOTE) is 0.71, a bit lower than the other two.

```
# get auc value
roc_predict <- predict(rf1, newdata = testSplit, na.rm = TRUE, type = "prob")
auc <- roc(testSplit$loan_quality, roc_predict[, 2])

# plot roc curve
plot(auc, main = "ROC Curve for Random Forest", col = 2, lwd = 2)
```

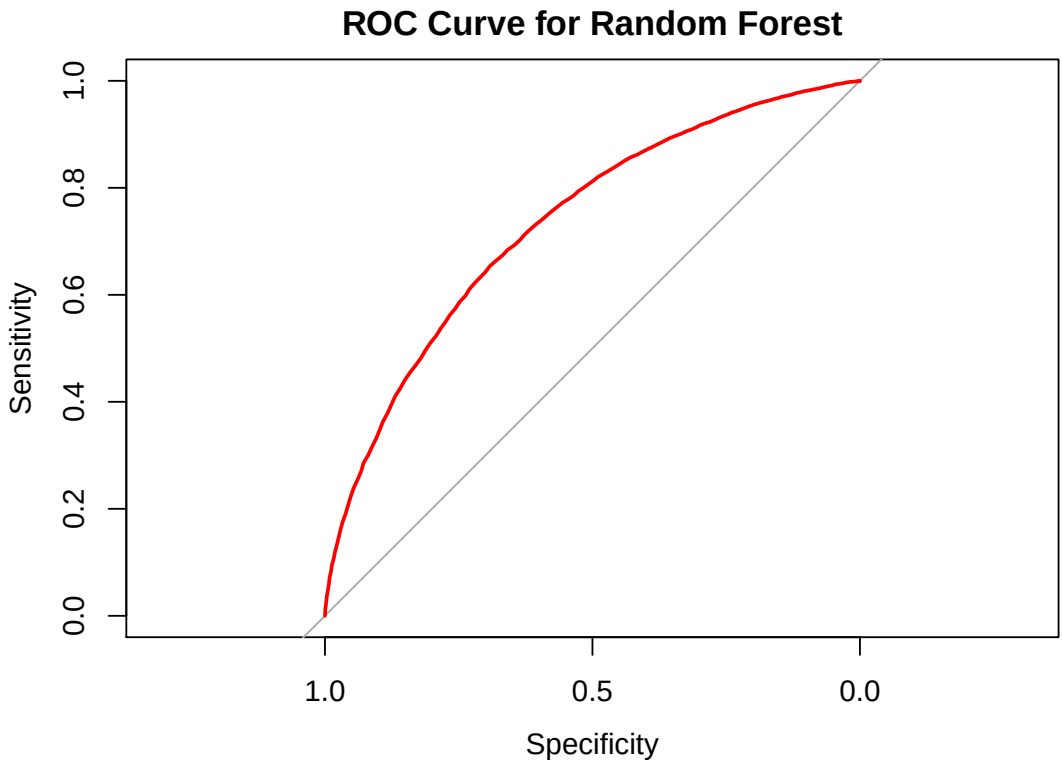


Figure 10: ROC Curve for regular random forest

```
print(auc)
```

```
Call:
roc.default(response = testSplit$loan_quality, predictor = roc_predict[,     2])

Data: roc_predict[, 2] in 8553 controls (testSplit$loan_quality Bad) < 36323 cases (t
Area under the curve: 0.7329
```

```
# get auc value
roc_predict_smote <- predict(rf2, testSplit, na.rm = TRUE, type = "prob")
auc_smote <- roc(testSplit$loan_quality, roc_predict_smote[, 2])

# plot roc curve
plot(auc_smote, main = "ROC Curve for Random Forest with SMOTE", col = 2,
    lwd = 2)
```

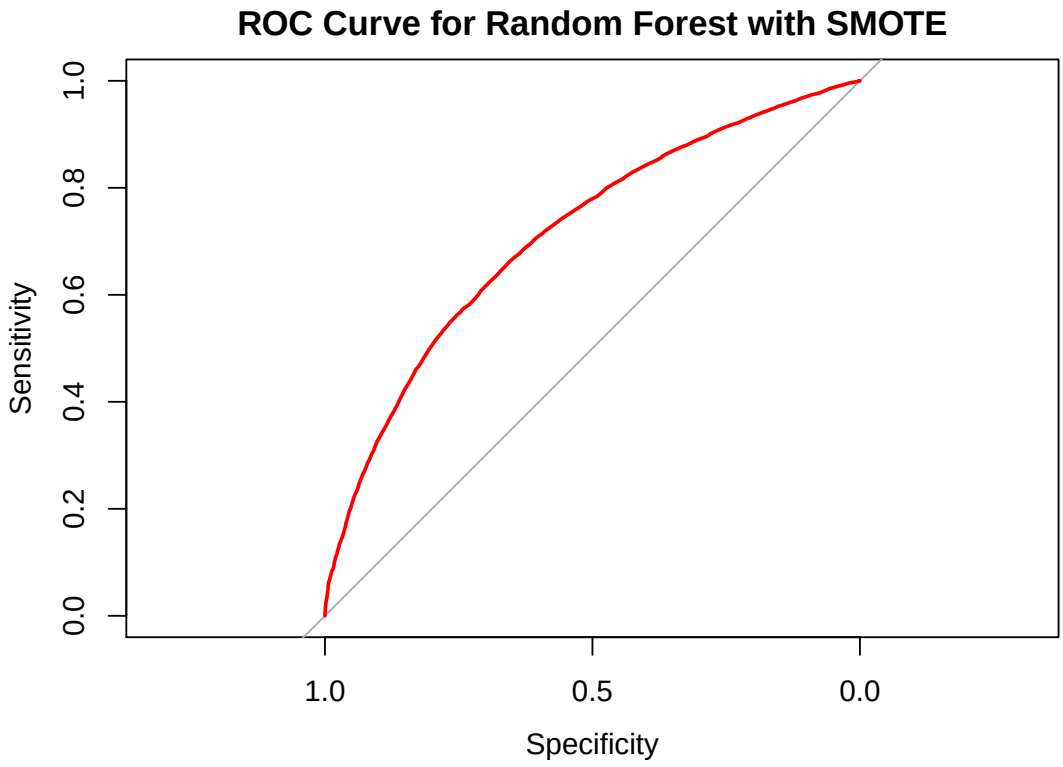


Figure 11: ROC Curve for random forest with SMOTE

```
print(auc_smote)
```

```
Call:
roc.default(response = testSplit$loan_quality, predictor = roc_predict_smote[,     2]

Data: roc_predict_smote[, 2] in 8553 controls (testSplit$loan_quality Bad) < 36323 ca
Area under the curve: 0.7124
```

```
# get auc value for logistic regression
auc_logit <- roc(testSplit$loan_quality, glm.probs)

# plot roc curve
plot(auc_logit,main="ROC Curve for Logistic Regression",col=2,lwd=2)
```

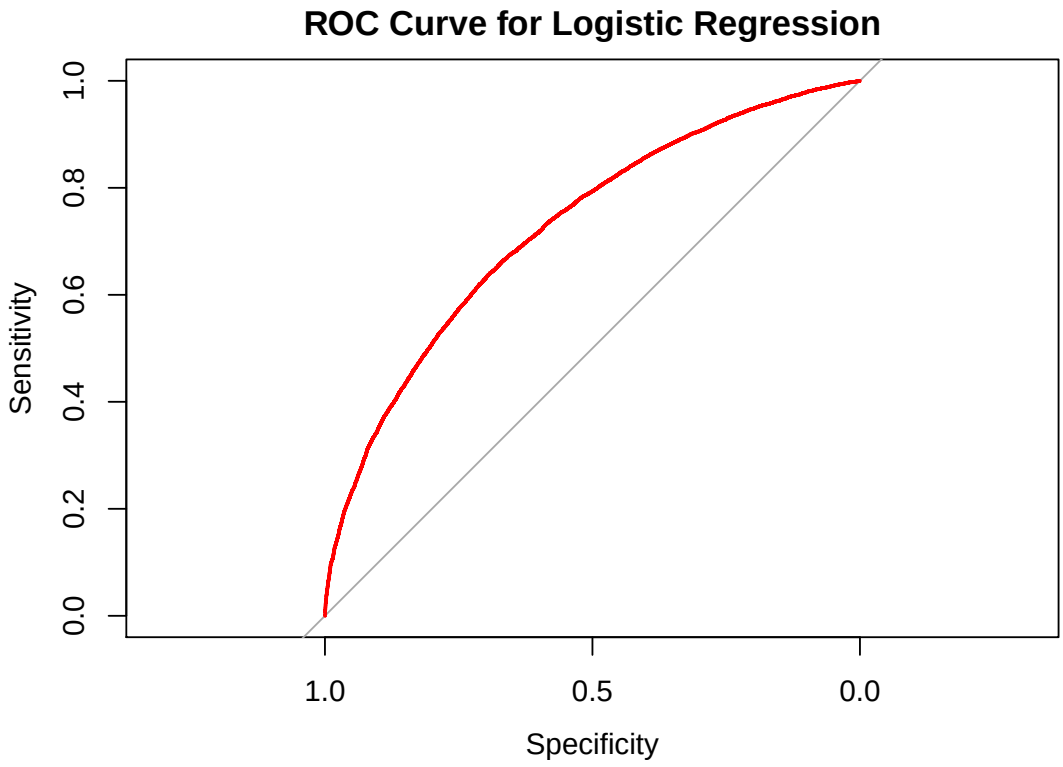


Figure 12: ROC Curve for logistic regression

```
print(auc_logit)
```

```
Call:
roc.default(response = testSplit$loan_quality, predictor = glm.probs)

Data: glm.probs in 8553 controls (testSplit$loan_quality Bad) < 36323 cases (testSpli
Area under the curve: 0.725
```

7.4 Finding the Best Cutoff

Now that we have the ROC curves, we can make an informed decision as to which cutoff we might want to use. A method available in the pROC library is finding the cutoff that is closest to the top left corner. We will run rf1's ROC again, with only a subset of the test set (which we will call evaluating set), so that this set is independent from the test set. Then we will run prediction with the best cutoff using the rest of the test subset. See below in Table 5 bad loans are predicted approximately 70%

of the time, while 35% of good loans are considered bad - an improvement over the original models.

```
# create evaluating test set and rest
set.seed(1341)
indexes <- sample(nrow(testSplit), 10000)
evaluating_set <- testSplit[indexes, ]
test_set_subset <- testSplit[-indexes, ]

# run rf1 ROC on evaluating set
roc_predict <- predict(rf1, newdata = evaluating_set, na.rm = TRUE, type = "prob")
auc <- roc(evaluating_set$loan_quality, roc_predict[, 2])

# get 'best' cutoff value
rf1_threshold <- coords(auc, x = "best", best.method = "closest.topleft")
rf1_threshold
```

```
  threshold specificity sensitivity
  0.8150000   0.6707946   0.6577772
```

```
# predict test_set_subset results with cutoff value
subset_predict <- predict(rf1, newdata = test_set_subset, na.rm = TRUE,
    type = "prob")
best_cutoff_predictions <- factor(ifelse(subset_predict[, 2] > rf1_threshold[1],
    "Good", "Bad"))

kable(table(test_set_subset$loan_quality, best_cutoff_predictions), caption = "Confus
    label = "bestcutoff") %>% kable_styling(latex_options = c("hold_position"))
```

Table 5: Confusion Matrix using "best" cutoff threshold

	Bad	Good
Bad	4520	2095
Good	9464	18797

8 K-fold Cross Validation

Another important topic when tuning machine learning models is k-fold cross validation. Notice that when we created training and testing datasets, we are only fitting the training data to generate the model. This means we are losing the information in the testing dataset when it is separated. K-fold cross validation deals with this problem. This cross validation method divides the data sets into k splits, and tests it k times - every split is held as the testing dataset once. Then, it averages the error rate to arrive

at an accuracy figure. This process can be repeated as many times as requested (in the code below it was repeated twice) to increase reliability in predicting accuracy. To ensure the model is trained correctly, it is important that in each split the proportion of good and bad loans is the same, otherwise there might be a chunk in which there are no defaulted loans. This is called stratified cross validation.[7]

One additional feature in this method is that it allows for the optimization of hyperparameters (such as mtry, ntree, etc.), by running the algorithm for a specified amount of times (in the code below, 4 times for mtry). Bear in mind that rerunning the code so much may cause it to be very slow, and take hours (sometimes days!) to run. That's why I used the doSNOW package, which specifies how many threads should be used while running the code. Since my computer has 8 threads, I set 6 threads for this task, leaving some space for other applications (if necessary). See the code below for results:

```
# Create 15 total folds to maintain default ratio.
set.seed(2346)
cv.10.folds <- createMultiFolds(trainSplit$loan_quality, k = 4, times = 2)

# Set up caret's trainControl object
# 4- fold validation repeated 2x, parameter search specified (not random)
ctrl.1 <- trainControl(method = "repeatedcv", number = 4,
                       repeats = 2, index = cv.10.folds,
                       classProbs = TRUE,
                       summaryFunction = twoClassSummary,
                       search = "grid")


# doSNOW package on multi-core training, since we're going
# to be training a lot of trees.
cl <- makeCluster(6, type = "SOCK")
registerDoSNOW(cl)


# Set seed for reproducibility and train (ROC as performance measure)
# Testing lower mtry values since splits with all variables make no sense
set.seed(1234)
rf4.cv1 <- train(x = rf_predictors, y = trainSplit$loan_quality,
                 method = "rf", tuneGrid = expand.grid(mtry = c(2, 4, 6, 8)),
                 ntree = 300, trControl = ctrl.1,
                 metric = "ROC")
```

[7] Leek (2013) is a very good introduction to cross validation.

```r
# Shutdown cluster
stopCluster(cl)
remove(cl)
registerDoSEQ()

# Check out results
rf4.cv1
```

```
Random Forest

104850 samples
    25 predictor
     2 classes: 'Bad', 'Good'

No pre-processing
Resampling: Cross-Validated (4 fold, repeated 2 times)
Summary of sample sizes: 78637, 78638, 78638, 78637, 78638, 78637, ...
Resampling results across tuning parameters:

  mtry  ROC        Sens         Spec
  2     0.6734651  0.007315255  0.9992760
  4     0.6731614  0.018172212  0.9976232
  6     0.7005898  0.029279921  0.9956894
  8     0.6993197  0.036458007  0.9941379

ROC was used to select the optimal model using the largest value.
The final value used for the model was mtry = 6.
```

```r
plot(rf4.cv1)
```

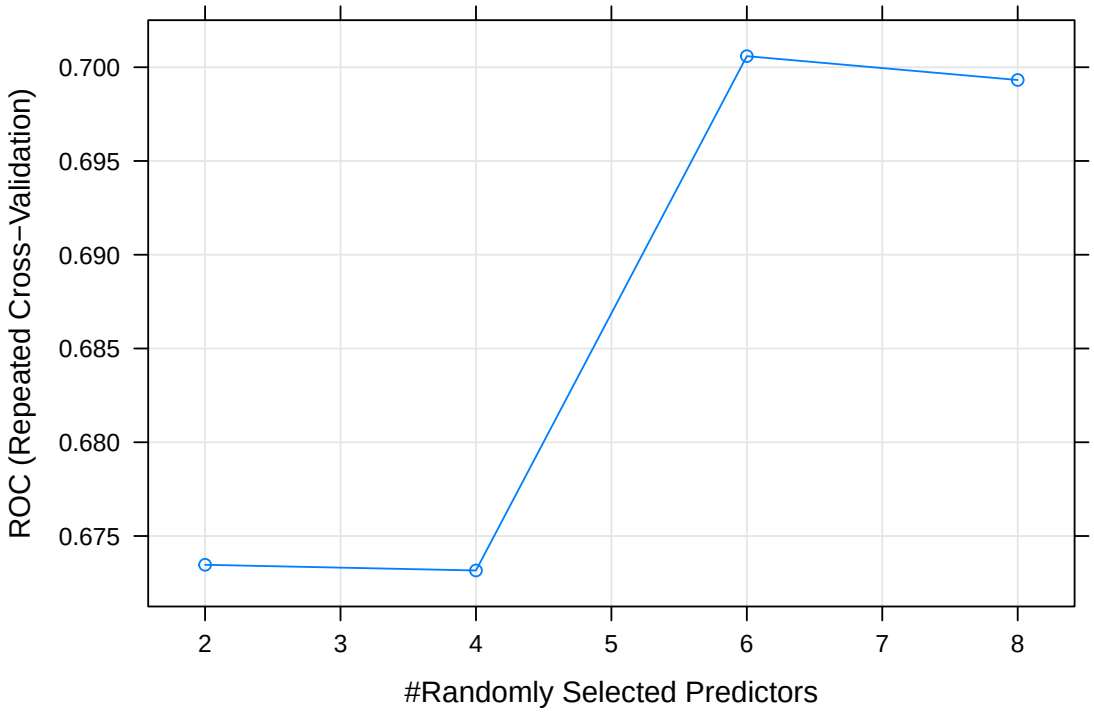


Figure 13: AUC for different mtry values

```
# roc/auc in test data
bad_loan.rf.pr_cv <- predict(rf4.cv1, testSplit, na.rm = TRUE, type = 'prob')
auc3 <- roc(testSplit$loan_quality, bad_loan.rf.pr_cv[,2])

# Plot roc curve
plot(auc3,main="ROC Curve with K-Fold Cross Validation",col=2,lwd=2)
```

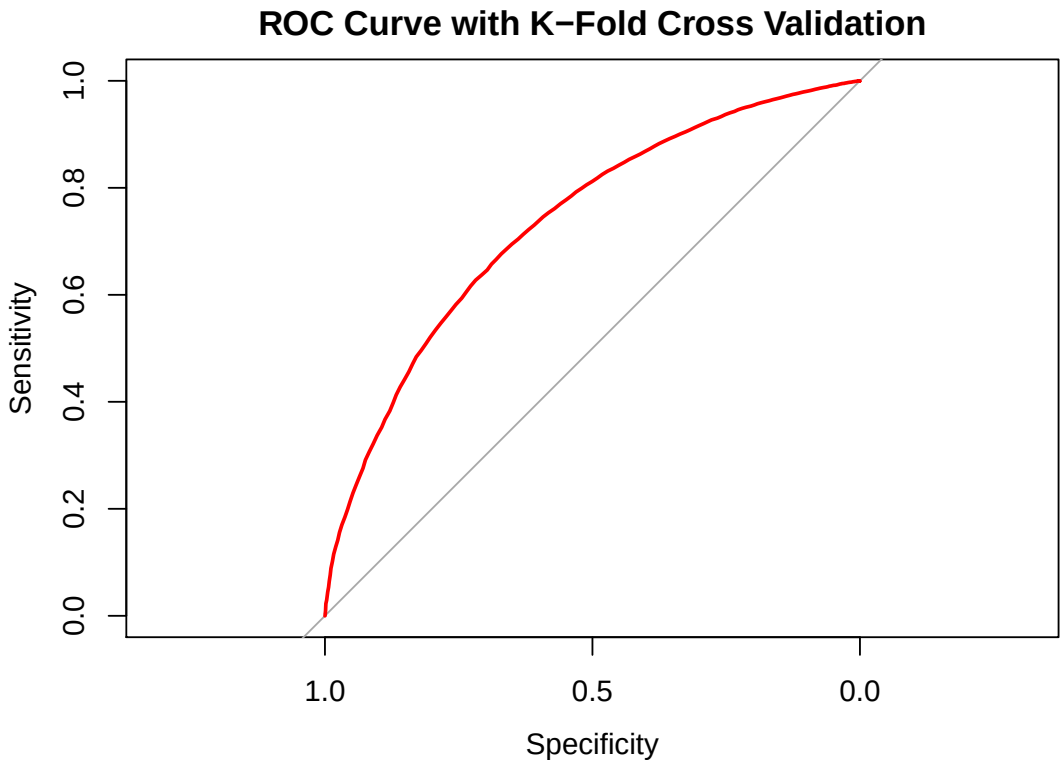


Figure 14: ROC Curve for 4-fold Cross Validation

```
print(auc3)
```

```
Call:
roc.default(response = testSplit$loan_quality, predictor = bad_loan.rf.pr_cv[,     2]

Data: bad_loan.rf.pr_cv[, 2] in 8553 controls (testSplit$loan_quality Bad) < 36323 ca
Area under the curve: 0.7341
```

With testing, we see that the best AUC results occur with mtry = 6, when compared to 2, 6 and 8. The difference is subtle- at most 0.03. However, the values for sensivity are very low, which means the model is assuming almost everything to be a good loan, and ignoring the bad ones. In this scenario, we achieve an AUC of .73 with the test data, which is only slightly better than our earlier test.

9 Neural Networks

9.1 Overview

Neural networks are another nonlinear model that can be applied in machine learning. The basic structure of the model follows like this:

1) Linear combinations of the predictors are taken to form what is called hidden units. In Figure 15, the predictors (the inputs) are I1 and I2, while the hidden units are H1 and H2.

2) Each hidden unit goes through a transformation (usually logistic) so that the results are between 0 and 1.

3) Finally, a linear combination of the hidden values are taken to arrive at the outcome (in case of a regression), or a value $f_{il}(x)$ (for classification), which means it is the predicted probability of the outcome, as a function of x (the predictors) in the i^{th} sample and the l^{th} class. In Figure 15, the outcome is O1.

The model above would have one layer of hidden units (data flows in the order input→hidden unit→output), which, in most cases, is enough. The other possibility would be input→hidden unit→hidden unit→output, for a two-layer neural network, but that is not necessary in our case. Figure 15 shows a neural network with 2 inputs and 2 neurons in the hidden layer (B1 and B2 stand for the intercepts in each layer):

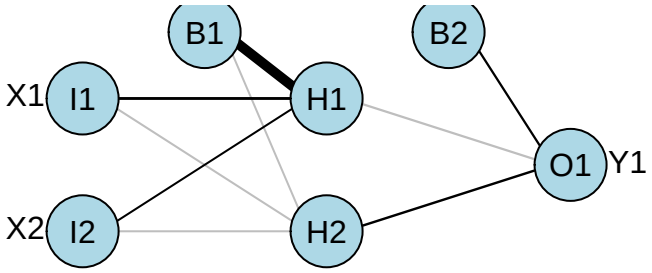


Figure 15: Neural network graphical example

Below, we are running a simple neural network, with only 3 neurons in the hidden layer. We then predict which loans will default on the test dataset, using this model.

```
# 1st neural net: unscaled data, only 3 neurons
set.seed(1492)
nn1 <- nnet(loan_quality ~ .,
            data = trainSplit,
            size = 3,
            trace = FALSE)

# show table with results
kable(table(testSplit$loan_quality,
            predict(nn1, testSplit, type = "class")))
```

	Good
Bad	8553
Good	36323

```
# predict results on test data and provide AUC
bad_loan.nn1.pr <- predict(nn1, testSplit, na.rm = TRUE, type = 'raw')
auc_nn1 <- roc(testSplit$loan_quality, bad_loan.nn1.pr)
print(auc_nn1$auc)
```

```
Area under the curve: 0.5358
```

9.2 Scaling

Notice in nn1 how the network predicts all loans to be good, and that the ROC is close 0.5 (a random predictor). Thus, this model adds almost no value to the analysis. This happened because the data was not normalized - data normalization is especially important for neural networks. So, for nn2, I scaled all numeric predictors to have mean 0 and standard deviation 1. This transformation makes the neural network much more effective in predicting defaults (ROC increased from 0.53 to 0.73).

```
# 2th neural net: scaled data
set.seed(1492)
nn2 <- nnet(loan_quality ~ .,
            data = scale.trainSplit,
            size = 3,
            trace = FALSE)

# show table
kable(table(scale.testSplit$loan_quality,
            predict(nn2, scale.testSplit, type = "class")))
```

	Bad	Good
Bad	926	7627
Good	782	35541

```
# predict results on test data and provide AUC
bad_loan.nn2.pr <- predict(nn2, scale.testSplit, na.rm = TRUE, type = 'raw')
auc_nn2 <- roc(scale.testSplit$loan_quality, bad_loan.nn2.pr)
print(auc_nn2$auc)
```

```
Area under the curve: 0.7292
```

9.3 Averaging

Since most problems are not linear, the model finds estimates that are only locally optimal (and the parameters have arbitrary initial values), meaning the estimates converge to the closest local minimum. So, we might have very different estimates with similar performance. In order to avoid this issue, several models are started with different values and the results are averaged. This method is called model averaging, and it is used below to optimize predictions. nn3 below uses this method.

```
# 3nd neural net: averaged neural nets for better optimization
set.seed(1492)
nn3 <- avNNet(loan_quality ~ .,
              data = scale.trainSplit,
              size = 3,
              trace = FALSE,
              entropy = TRUE, # explained in next section
              softmax = TRUE) # explained in next section

# show table
kable(table(scale.testSplit$loan_quality,
            predict(nn3, scale.testSplit, type = "class")))
```

	Bad	Good
Bad	705	7848
Good	480	35843

```
# predict results on test data and provide AUC
bad_loan.nn3.pr <- predict(nn3, scale.testSplit,
                           na.rm = TRUE, type = 'raw')
auc_nn3 <- roc(scale.testSplit$loan_quality, bad_loan.nn3.pr[,2])
print(auc_nn3$auc)
```

```
Area under the curve: 0.7395
```

9.4 Mathematical Background

In this section, we give the mathematical background to the neural networks we have been running.

In classification problems, there is an issue: the values for f(x) do not sum to 1. So, you take the softmax transformation to solve this and arrive at the probability of each class occurring for that sample. The transformation is as follows:

$$f_{il} = \frac{e^{f_{il}(x)}}{\sum l(e^{f_{il}(x)})}$$

Another question you may ask is: what do these neural networks optimize? The answer depends on if the problem is a regression or a classification one. For regression problems, the neural networks minimize the sum of squared errors:

$$\sum_{i=1}^{n}(y_i - f_i(x))^2$$

The sum of squared errors is minimized to ensure that the predictions are as close as possible to the training data values.

For classification problems, you also have the possibility to maximize the likelihood of a Bernoulli Distribution. The likelihood function is

$$\sum_{l=1}^{C}\sum_{i=1}^{n} y_{ii} * ln(f_{il}^{*}(x))$$

This function is called entropy. Every observation in the training set has the target variable set to 1 (in case the loan defaults) or 0 (in case it doesn't). This function basically tries to match the distribution in the training set the best it can, and it has more theoretical validity than the SSE approach[8]. In fact, most modern neural networks use this function[9]. Neural network functions usually default to the SSE method, but you can change this setting by adding entropy = TRUE in the code (as can be seen in nn5 below).

Finally, how many neurons should we fit in the model? If we fit too few, the model will underfit the data. Conversely, if we fit too many, the model will overfit. It might come down to trial and error, but Heaton (2005) indicates a good rule of thumb to start testing: two thirds of the size of the input layer plus the size of the output layer. In our dataset, there are 25 predictors and thus 25 input neurons, and this is a binary

[8] Kuhn and Johnson (2013), 333-334

[9] see Goodfellow and Courville (2016) and Brownlee (2019) for more information

classification problem ("good" and "bad") and thus we have 1 output neuron. This would give 17 neurons. In order to save computation time, we will be using only 9 neurons to model our predictions. Also, we could potentially fit more than one hidden layer, but, according to Pacelli and Azzollini (2011), in most problems there is no reason for that. Note that the transformations and the change in the hidden layer size improved the ROC for the model.

```
# 5th neural net: 9 neurons in hidden layer
# Maximum number of weights increased to 2000 to acommodate 9 hidden units (default
set.seed(1492)
nn5 <- avNNet(loan_quality ~ .,
              data = scale.trainSplit,
              size = 9,
              MaxNWts = 2000,
              softmax = TRUE,    #include softmax transformation
              trace = FALSE,     #don't output optimization iterations
              entropy = TRUE)    #use entropy instead of least squares

# show table
kable(table(scale.testSplit$loan_quality,
            predict(nn5, scale.testSplit, type = "class")))
```

	Bad	Good
Bad	1110	7443
Good	822	35501

```
# predict results on test data and provide AUC
bad_loan.nn5.pr <- predict(nn5, scale.testSplit, na.rm = TRUE, type = 'raw')
auc_nn5 <- roc(scale.testSplit$loan_quality, bad_loan.nn5.pr[,2])
print(auc_nn5$auc)
```

```
Area under the curve: 0.7491
```

9.5 Pre-processing Techniques

Other possibility to increase your model's accuracy is through pre-processing techniques. We have already mentioned scaling (which is a prerequisite to running neural network models), but there are other possibilities, such as centering the data (all values around 0) and the spatial-sign transformation, which projects the data onto the unit circle of the p predictors - so that outliers are brought to the same unit as non-outliers. For this transformation, the data must be centered and scaled.

```
# 6th neural net: pre-processing techniques

set.seed(1492)
```

```
nnet6 <- train(loan_quality ~ ., data = scale.trainSplit,
               method = "avNNet",
               verbose = FALSE,
               trace = FALSE,
               metric = "ROC",
               tuneGrid = data.frame(size = 9, decay = 0, bag = F),
               preProc = c("center", "scale", "spatialSign"),
               MaxNWts = 2000,
               allowParallel = TRUE,
               softmax = TRUE,
               entropy = TRUE,
               trControl = trainControl(method = "none",
                                        classProbs = TRUE,
                                        summaryFunction = twoClassSummary))

# show table
kable(table(scale.testSplit$loan_quality,
            predict(nnet6, scale.testSplit, type = "raw")))
```

	Bad	Good
Bad	256	8297
Good	145	36178

```
# predict results on test data and provide AUC
bad_loan.nn6.pr <- predict(nnet6, scale.testSplit, na.rm = TRUE, type = 'prob')
auc_nn6 <- roc(scale.testSplit$loan_quality, bad_loan.nn6.pr[,2])
print(auc_nn6$auc)
```

```
Area under the curve: 0.7435
```

9.6 Decay and Cross Validation

To avoid overfitting, we can use set a weight decay. This weight decay (usually denoted by λ) penalizes large coefficients on the predictors, so that the predictions become "smoother" and capture less noise. Below is a formula of the sum of squared errors with weight decay:

$$\sum_{i=1}^{n}(y_i - f_i(x))^2 + \lambda * \sum_{k=1}^{H}\sum_{j=0}^{P}\beta_{jk}^2 + \lambda * \sum_{k=0}^{H}\gamma_k^2$$

β_{jk} are the coefficients in the initial regression from input to hidden unit and γ_k are the coefficients from hidden unit to output. So, increasing lambda reduces overfitting.

Lambda is a parameter chosen by the user, and it usually ranges between 0 and 2.[10]

```
# set up CV
cv.9.folds <- createMultiFolds(trainSplit$loan_quality, k = 3, times = 3)
fitControl <- trainControl(method = "repeatedcv", number = 3,
                           repeats = 3, classProbs = TRUE,
                           summaryFunction = twoClassSummary,
                           index = cv.9.folds, search = "grid")

# comparison with weight decays and sizes
nnetGrid <-  expand.grid(size = c(3, 8, 13), decay = c(0,0.2,1,2))

cl <- makeCluster(6, type = "SOCK")
registerDoSNOW(cl)

set.seed(825)
nnet_fit <- train(loan_quality ~ ., data = scale.trainSplit,
                  method = "nnet",
                  trControl = fitControl,
                  verbose = FALSE,
                  trace = FALSE,
                  tuneGrid = nnetGrid,
                  metric = "ROC",
                  allowParallel = TRUE,
                  MaxNWts = 2000)
stopCluster(cl)
remove(cl)
registerDoSEQ()
```

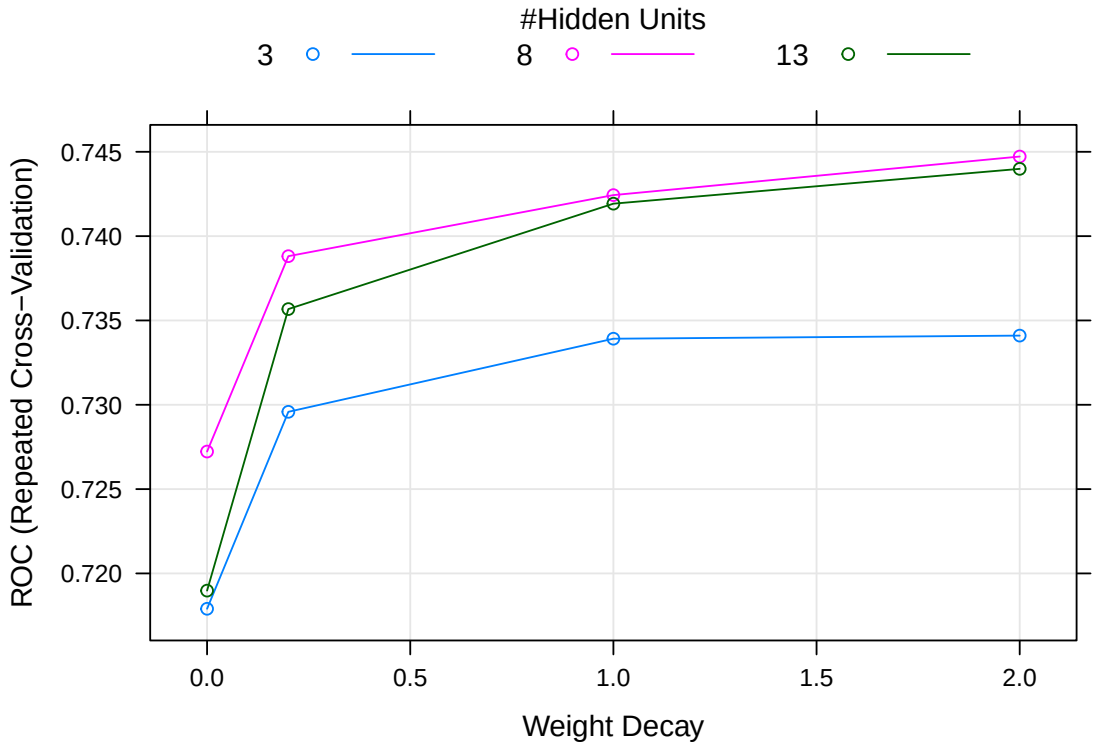
```
nnet_fit
```

```
Neural Network

104850 samples
    25 predictor
     2 classes: 'Bad', 'Good'

No pre-processing
Resampling: Cross-Validated (3 fold, repeated 3 times)
Summary of sample sizes: 69900, 69901, 69899, 69901, 69900, 69899, ...
Resampling results across tuning parameters:
```

[10] See Kuhn and Johnson (2013) pages 141-145 for more details.

```
 size  decay  ROC        Sens        Spec
  3    0.0    0.7179024  0.02164739  0.9948450
  3    0.2    0.7295825  0.06022387  0.9885529
  3    1.0    0.7339173  0.06075426  0.9884230
  3    2.0    0.7341020  0.02745601  0.9955376
  8    0.0    0.7272262  0.14846964  0.9659108
  8    0.2    0.7388116  0.14055893  0.9720809
  8    1.0    0.7424332  0.11932557  0.9773578
  8    2.0    0.7447204  0.11876233  0.9783337
 13    0.0    0.7189807  0.16659216  0.9569428
 13    0.2    0.7356749  0.15004200  0.9684292
 13    1.0    0.7419253  0.14289268  0.9722973
 13    2.0    0.7439889  0.13579250  0.9744773

ROC was used to select the optimal model using the largest value.
The final values used for the model were size = 8 and decay = 2.
```

```r
plot(nnet_fit, metric = "ROC")
```



9.7 Best Cutoff Value

Using the cross validation model, let's predict loans using the best cutoff value for nn5 and check the results. Table 6 shows the results, with 67% of bad loans being caught and only 30% of good loans incorrectly considered bad.

```
# create evaluating test set and subset
# of test with remaining observations
set.seed(129)
indexes <- sample(nrow(scale.testSplit),
    10000)
evaluating_set <- scale.testSplit[indexes,
    ]
test_set_subset <- scale.testSplit[-indexes,
    ]

# run rf1 ROC on evaluating set
roc_predict <- predict(nn5, newdata = evaluating_set,
    na.rm = TRUE, type = "prob")
auc <- roc(evaluating_set$loan_quality, roc_predict[,
    2])

# get 'best' cutoff value
nnet_threshold <- coords(auc, x = "best",
    best.method = "closest.topleft")
nnet_threshold
```

```
  threshold specificity sensitivity
  0.7971699   0.6719745   0.6933218
```

```
# predict test_set_subset results with
# cutoff value
subset_predict <- predict(nn5, newdata = test_set_subset,
    na.rm = TRUE, type = "prob")
best_cutoff_predictions <- factor(ifelse(subset_predict[,
    2] > nnet_threshold[1], "Good", "Bad"))

kable(table(test_set_subset$loan_quality,
    best_cutoff_predictions), caption = "Confusion Matrix using \"best\" cutoff thres
    label = "bestcutoffnnet") %>% kable_styling(latex_options = c("hold_position"))
```

Table 6: Confusion Matrix using "best" cutoff threshold

	Bad	Good
Bad	4476	2193
Good	8627	19580

10 Conclusion

Machine learning can be a powerful tool for many applications, and we have just seen one interesting application in bank loans. We have also seen that there are many approaches to build your model, and each requires a different way to prepare the data. So, don‘t forget this: data cleaning is very important! Finally, with this toolset, you are able to start building complex models on your own.

11 References

Beque, A., and S. Lessman. 2017. "Extreme Learning Machines for Credit Scoring: An Empirical Evaluation." *Expert Systems with Applications* 86: 42–53.

Brownlee, Jason. 2019. "Loss and Loss Functions for Training Deep Learning Neural Networks." 2019. https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning

Chawla, Bowyer, N., and P. Kegelmeyer. 2002. "SMOTE: Synthetic Minority over-Sampling Technique." *Journal of Artificial Intelligence Research* 16: 321–57.

Crone, Lessman, S., and R. Stahlbock. 2005. "The Impact of Preprocessing on Data Mining: An Evaluation of Classifier Sensitivity in Direct Marketing." *European Journal of Operational Research* 173: 781–800.

Goodfellow, Bengio, I., and A. Courville. 2016. *Deep Learning*.

Heaton, J. 2005. *Programming Neural Networks in Java*.

James, Witten, G., and R. Tibshiran. 2013. *An Introduction to Statistical Learning*. Springer Science+Business Media, Inc.

Kuhn, M., and K. Johnson. 2013. *Applied Predictive Modeling*. Springer Science+Business Media, Inc.

Leek, Jeff. 2013. "CrossValidation." Youtube. 2013. https://www.youtube.com/watch?v=CmEqvD_ov2o.

Pacelli, V., and M. Azzollini. 2011. "An Artificial Neural Network Approach for Credit Risk Management." *Journal of Intelligent Learning Systems and Applications* 3: 103–12.

Polena, Michal. 2017. "Performance Analysis of Credit Scoring Models on Lending Club Data."

Provost, F., and G. Weiss. 2003. "Learning When Training Data Are Costly: The Effect of Class Distribution on Tree Induction." *Journal of Artificial Intelligence Research* 19: 315–54.

12 Appendix A: Predictor Variables Definitions

Below is a breakdown of the explanation for every variable provided in the dataset.

Variable	Description
addr_state	The state provided by the borrower in the loan application
annual_inc	The self-reported annual income provided by the borrower during registration.
annual_inc_joint	The combined self-reported annual income provided by the co-borrowers during registration
application_type	Indicates whether the loan is an individual application or a joint application with two co-borrowers
collection_recovery_fee	post charge off collection fee
collections_12_mths_ex_med	Number of collections in 12 months excluding medical collections
delinq_2yrs	The number of 30+ days past-due incidences of delinquency in the borrower's credit file for the past 2 years
desc	Loan description provided by the borrower
dti	A ratio calculated using the borrower's total monthly debt payments on the total debt obligations, excluding mortgage and the requested LC loan, divided by the borrower's self-reported monthly income.
dti_joint	A ratio calculated using the co-borrowers' total monthly payments on the total debt obligations, excluding mortgages and the requested LC loan, divided by the co-borrowers' combined self-reported monthly income
earliest_cr_line	The month the borrower's earliest reported credit line was opened
emp_length	Employment length in years. Possible values are between 0 and 10 where 0 means less than one year and 10 means ten or more years.
emp_title	The job title supplied by the Borrower when applying for the loan.*
funded_amnt	The total amount committed to that loan at that point in time.
funded_amnt_inv	The total amount committed by investors for that loan at that point in time.
grade	LC assigned loan grade

home_ownership	The home ownership status provided by the borrower during registration. Our values are: RENT, OWN, MORTGAGE, OTHER.
id	A unique LC assigned ID for the loan listing.
initial_list_status	The initial listing status of the loan. Possible values are – W, F
inq_last_6mths	The number of inquiries in past 6 months (excluding auto and mortgage inquiries)
installment	The monthly payment owed by the borrower if the loan originates.
int_rate	Interest Rate on the loan
issue_d	The month which the loan was funded
last_credit_pull_d	The most recent month LC pulled credit for this loan
last_pymnt_amnt	Last total payment amount received
last_pymnt_d	Last month payment was received
loan_amnt	The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.
loan_status	Current status of the loan
member_id	A unique LC assigned Id for the borrower member.
mths_since_last_delinq	The number of months since the borrower's last delinquency.
mths_since_last_major_derog	Months since most recent 90-day or worse rating
mths_since_last_record	The number of months since the last public record.
next_pymnt_d	Next scheduled payment date
open_acc	The number of open credit lines in the borrower's credit file.
out_prncp	Remaining outstanding principal for total amount funded
out_prncp_inv	Remaining outstanding principal for portion of total amount funded by investors
policy_code	publicly available policy_code=1, new products not publicly available policy_code=2
pub_rec	Number of derogatory public records
purpose	A category provided by the borrower for the loan request.
pymnt_plan	Indicates if a payment plan has been put in place for the loan
recoveries	post charge off gross recovery
revol_bal	Total credit revolving balance

revol_util	Revolving line utilization rate, or the amount of credit the borrower is using relative to all available revolving credit.
sub_grade	LC assigned loan subgrade
term	The number of payments on the loan. Values are in months and can be either 36 or 60.
title	The loan title provided by the borrower
total_acc	The total number of credit lines currently in the borrower's credit file
total_pymnt	Payments received to date for total amount funded
total_pymnt_inv	Payments received to date for portion of total amount funded by investors
total_rec_int	Interest received to date
total_rec_late_fee	Late fees received to date
total_rec_prncp	Principal received to date
url	URL for the LC page with listing data.
zip_code	The first 3 numbers of the zip code provided by the borrower in the loan application.
open_acc_6m	Number of open trades in last 6 months
open_il_6m	Number of currently active installment trades
open_il_12m	Number of installment accounts opened in past 12 months
open_il_24m	Number of installment accounts opened in past 24 months
mths_since_rcnt_il	Months since most recent installment accounts opened
total_bal_il	Total current balance of all installment accounts
il_util	Ratio of total current balance to high credit/credit limit on all install acct
open_rv_12m	Number of revolving trades opened in past 12 months
open_rv_24m	Number of revolving trades opened in past 24 months
max_bal_bc	Maximum current balance owed on all revolving accounts
all_util	Balance to credit limit on all trades
total_rev_hi_lim	Total revolving high credit/credit limit
inq_fi	Number of personal finance inquiries
total_cu_tl	Number of finance trades
inq_last_12m	Number of credit inquiries in past 12 months
acc_now_delinq	The number of accounts on which the borrower is now delinquent.
tot_coll_amt	Total collection amounts ever owed
tot_cur_bal	Total current balance of all accounts

13 Appendix B: List of Final Predictor Variables

Below is a breakdown of the model variables.

13.1 Original Variables Description

	Variable	Description
1	addr_state	The state provided by the borrower in the loan application
2	annual_inc	The self-reported annual income provided by the borrower during registration.
7	delinq_2yrs	The number of 30+ days past-due incidences of delinquency in the borrower's credit file for the past 2 years
9	dti	A ratio calculated using the borrower's total monthly debt payments on the total debt obligations, excluding mortgage and the requested LC loan, divided by the borrower's self-reported monthly income.
12	emp_length	Employment length in years. Possible values are between 0 and 10 where 0 means less than one year and 10 means ten or more years.
17	home_ownership	The home ownership status provided by the borrower during registration. Our values are: RENT, OWN, MORTGAGE, OTHER.
19	initial_list_status	The initial listing status of the loan. Possible values are – W, F
20	inq_last_6mths	The number of inquiries in past 6 months (excluding auto and mortgage inquiries)
21	installment	The monthly payment owed by the borrower if the loan originates.
34	open_acc	The number of open credit lines in the borrower's credit file.
38	pub_rec	Number of derogatory public records
39	purpose	A category provided by the borrower for the loan request.
42	revol_bal	Total credit revolving balance
43	revol_util	Revolving line utilization rate, or the amount of credit the borrower is using relative to all available revolving credit.
44	sub_grade	LC assigned loan subgrade
45	term	The number of payments on the loan. Values are in months and can be either 36 or 60.

47	total_acc	The total number of credit lines currently in the borrower's credit file
66	total_rev_hi_lim	Total revolving high credit/credit limit
72	tot_cur_bal	Total current balance of all accounts

13.2 Created Variables Description

Below is an explanation for every created variable added to the model.

Variable	Description
loan_quality	Target variable: displays whether a loan is considered good or bad
loan_income_ratio	Ratio between borrowers‘ annual income and total loan amount
last_credit_pull_d_v2	The most recent year LC pulled credit for this loan
year	Year loan was taken
credit_line_age	Number of years borrower has had a credit line
desc_length	Length of explanation description

14 Appendix C: Removed Variables Rationale

Below is a breakdown of the reason to remove columns. As can be seen, most have been removed because the data is almost homogenous or is unavailable at the time of underwriting:

Dropped Variable	Description
application_type	254,189 individual applications and 1 joint
collection_recovery_fee	Unknown variable at loan underwriting
collections	Unknown variable at loan underwriting
collections_12_mths_ex_med	Unknown variable at loan underwriting
desc	Too convoluted
earliest_cr_line	Transformed
emp_title	Too convoluted
funded_amnt	Unknown variable at loan underwriting
funded_amnt_inv	Unknown variable at loan underwriting
grade	Dominated by sub_grade
id	Randomly assigned to borrower
int_rate	Trying to assign int_rate as objective of analysis
issue_d	Transformed (too many outcomes to be considered factor)
last_credit_pull_d	Unknown variable at loan underwriting

last_pymnt_amnt	Unknown variable at loan underwriting
last_pymnt_d	Unknown variable at loan underwriting
loan_status	Transformed
member_id	Randomly assigned to borrower
next_pymnt_d	Unknown variable at loan underwriting
out_prncp	Unknown variable at loan underwriting
out_prncp_inv	Unknown variable at loan underwriting
policy_code	Every observation = 1
pymnt_plan	254,188 n and 2 y
recoveries	Unknown variable at loan underwriting
title	Too convoluted
total_pymnt	Unknown variable at loan underwriting
total_pymnt_inv	Unknown variable at loan underwriting
total_rec_int	Unknown variable at loan underwriting
total_rec_late_fee	Unknown variable at loan underwriting
total_rec_prncp	Unknown variable at loan underwriting
url	Too convoluted
verified_status_joint	254,189 N/A and 1 Source Verified
zip_code	Too convoluted
tot_coll_amt	Unknown variable at loan underwriting