Detecting Credit Card Fraud

An in-depth look into linear model extensions

Moritz C. Schaaf

2020-06-23

Contents

Introduction

Our understanding of shopping is changing rapidly. In the past years online shopping has become more and more omnipresent. E-commerce sales worldwide exhibit staggering growth rates of 20 percent annualy. However, the total sales are estimated to grow by only 5 percent annualy[1]. It is especially obvious this year: Due to the COVID-19 policies applied everywhere, the world moves away from in-store purcases. This rapidly changing business environment, however, also yields its perils. Fraudsters and malware becomes more subtle and the need for physical interaction with the victim futher decreases. One strategy to cope with this development is machine learning.

Therefore, this report will focus on machine learning algorithms to detect fraudulent credit card payments. The FBI[2] defines credit card fraud as *"the unauthorized use of a credit or debit card, or similar payment tool (ACH, EFT, recurring charge, etc.), to fraudulently obtain money or property".* This definition does not only include fraudulent e-commerce transactions but also physical unauthorized charges such as a family member using the card without permission. This report is part of the HarvardX Data Science Series[3] capstone project. The R and R Markdown codes are also available on my GitHub[4].

We will use a public dataset provided by the Machine Learning Group at the Université Libre de Bruxelles (Brussels). The most recent version is 3.0, last updated 2018-03-23. The full dataset can be found on kaggle[5] under a open database license. Which kind of fraud occured is not further specified, only if the transaction was fraudulent or not.

After performing some basic inspection of the dataset, we will explore logistic regression in detail and compaire the results with other well-known machine learning algorithms. The performance of the best algorithm is then further discussed.

[1] https://www.emarketer.com/content/global-ecommerce-2019
[2] https://www.fbi.gov/scams-and-safety/common-scams-and-crimes/credit-card-fraud
[3] https://www.edx.org/professional-certificate/harvardx-data-science
[4] https://github.com/mc-schaaf
[5] https://www.kaggle.com/mlg-ulb/creditcardfraud

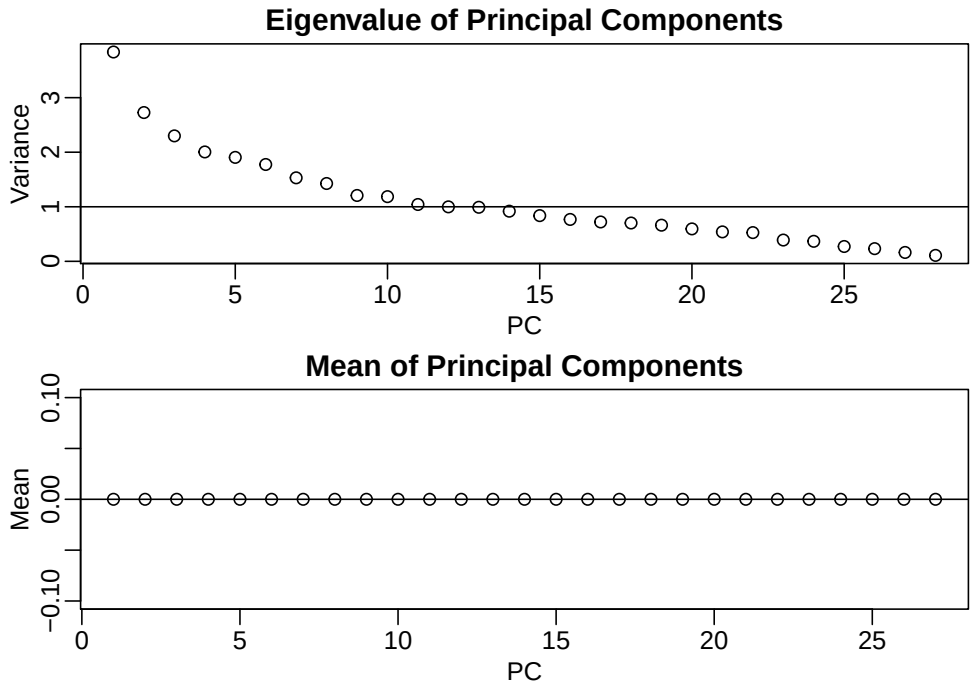
Dataset Preparation

The data was collected over two days in September 2013 and contains information about european cardholders. 492 transactions out of a total of 284807 transactions are fraudulent, leading to a highly unbalanced dataset where the positive cases (fraud) account for 0.173% of the cases. Due to privacy concerns most predictor variables are the first 28 Principal Components of a PCA. The only features not transformed are `Time` and `Amount`, where `Time` contains the seconds elapsed between each transaction and the first transaction in the dataset and `Amount` is the transaction amount. There are 0 missing values so this is a problem we don't have to deal with in this report.

Feature Engineering and Data Splitting

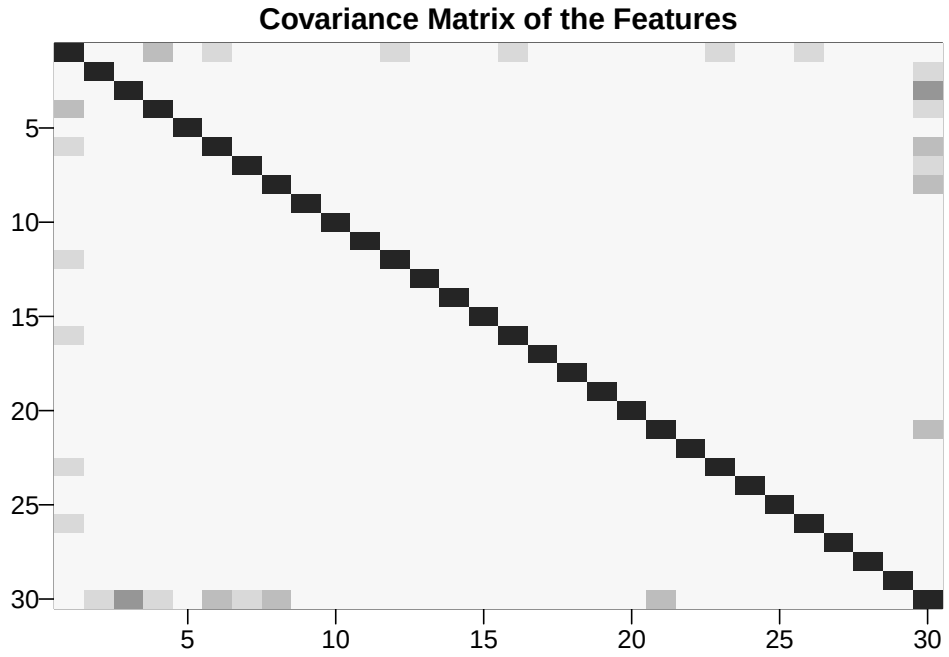
We will inspect the whole dataset before splitting into training, test and validation set. With this way of proceeding we can make sure we don't bias one of the subsets by accident. The Principal Components are named `V1`, `V2` ... `V28` and all have mean 0, as expected. Their *eigenvalue* has been preserved as their variance, respectively:

```
rafalib::mypar(2,1)
lapply(creditcard[,-c(1,30,31)], var) %>% unlist() %>% plot(xlab = "PC", ylab = "Variance")
abline(h = 1)
title("Eigenvalue of Principal Components")
lapply(creditcard[,-c(1,30,31)], mean) %>% unlist() %>% .[-c(1,30,31)] %>%
  plot(xlab = "PC", ylab = "Mean", ylim = c(-0.1,0.1))
abline(h = 0)
title("Mean of Principal Components")
```



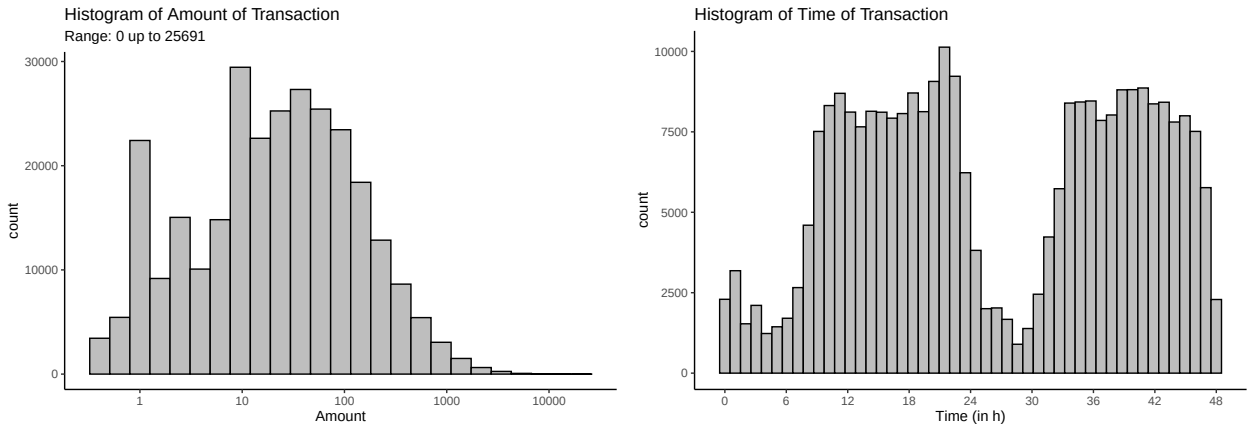
Following the definition of PCA, features `V1` up to `V28` are uncorrelated. Some small correlation with `Time` (first feature) and `Amount` (last feature) exists. However, this small a magnitude of covariance is not a problem for logistic regression or the other used machine learning algorithms.

```
rafalib::mypar(1,1)
rafalib::imagemat(abs(cor(creditcard[,-c(31)])), col = RColorBrewer::brewer.pal(7,"Greys"))
title("Covariance Matrix of the Features")
```



For `Amount` of Transaction we will transform the variable into a log10 variable. Thus, the influence of costly transactions (= outlieres) can be reduced, resulting in a higher performance of the algorithm. However, one could also decide against this approach as huge transactions should have a higher impact (= leverage) on the algorithm than small transactions. `Time` shows a bimodal distribution, as data was collected over a period of 48 hours and there are fewer activities over night. Here, a transformation reducing the time component to time of day will be implemented.

```
creditcard %>% ggplot() + geom_histogram(aes(x = ifelse(Amount < 0.5, 0.5, Amount)),
                                         color = "black", fill = "grey", bins = 25) +
  scale_x_continuous("Amount",trans = "log10") +
  ggtitle("Histogram of Amount of Transaction",
          subtitle = paste0("Range: ", range(creditcard$Amount)[1], " up to ",
                            round(range(creditcard$Amount)[2]))) +
  theme_classic()
creditcard %>% ggplot() + geom_histogram(aes(x = creditcard$Time/(60*60)),
                                         color = "black", fill = "grey", bins = 48) +
  scale_x_continuous("Time (in h)", breaks = 6*c(0:8)) +
  ggtitle("Histogram of Time of Transaction") + theme_classic()
```



None of the variables shows extreme outliers or scarce values that have to be considered for splitting the data. We will use two thirds of the data for training the algorithm. The remaining third will be split into a leaderboard set and final validation set of equal size. Note that the resulting esimates can be quite wiggly, as only 492 * 1/3 * 1/2 = 82 Fraud cases will be in each test set. On the other hand this approach will prevent hillclimbing with different algorithms on the leaderboard set and provides an rather unbiased estimate of real-world performance.

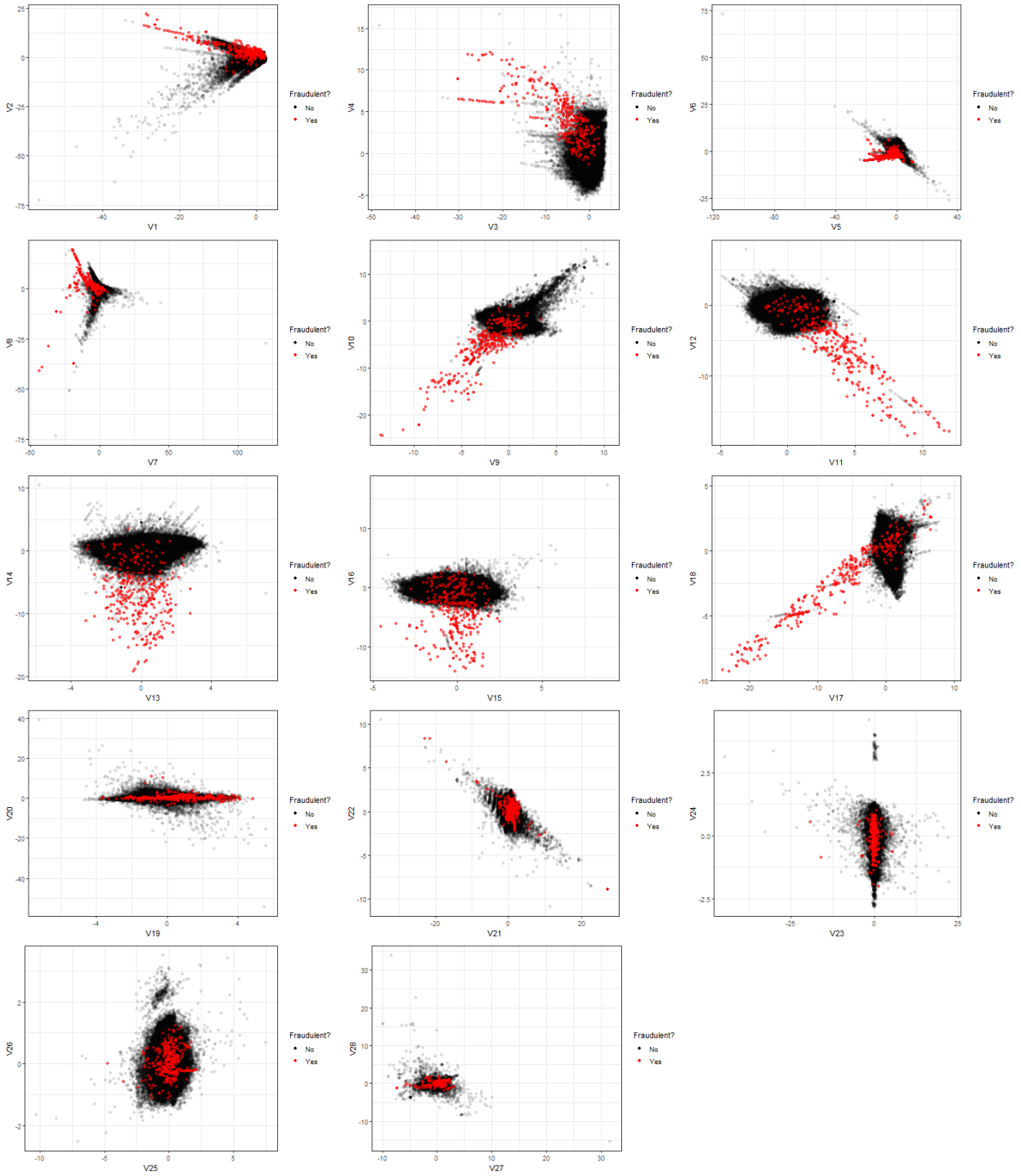
```
# Test set will be 33% of the data
set.seed(1, sample.kind="Rounding")
test_index <- createDataPartition(y = creditcard$Class, times = 1, p = 1/3, list = FALSE)
data_test <- creditcard[test_index,] %>% as.data.frame()
data_train <- creditcard[-test_index,] %>% as.data.frame()

# Split Test 50-50 again for final evaluation
test_index <- createDataPartition(y = data_test$Class, times = 1, p = 1/2, list = FALSE)
data_validation <- data_test[test_index,]
data_test <- data_test[-test_index,]
```

Data Visualization

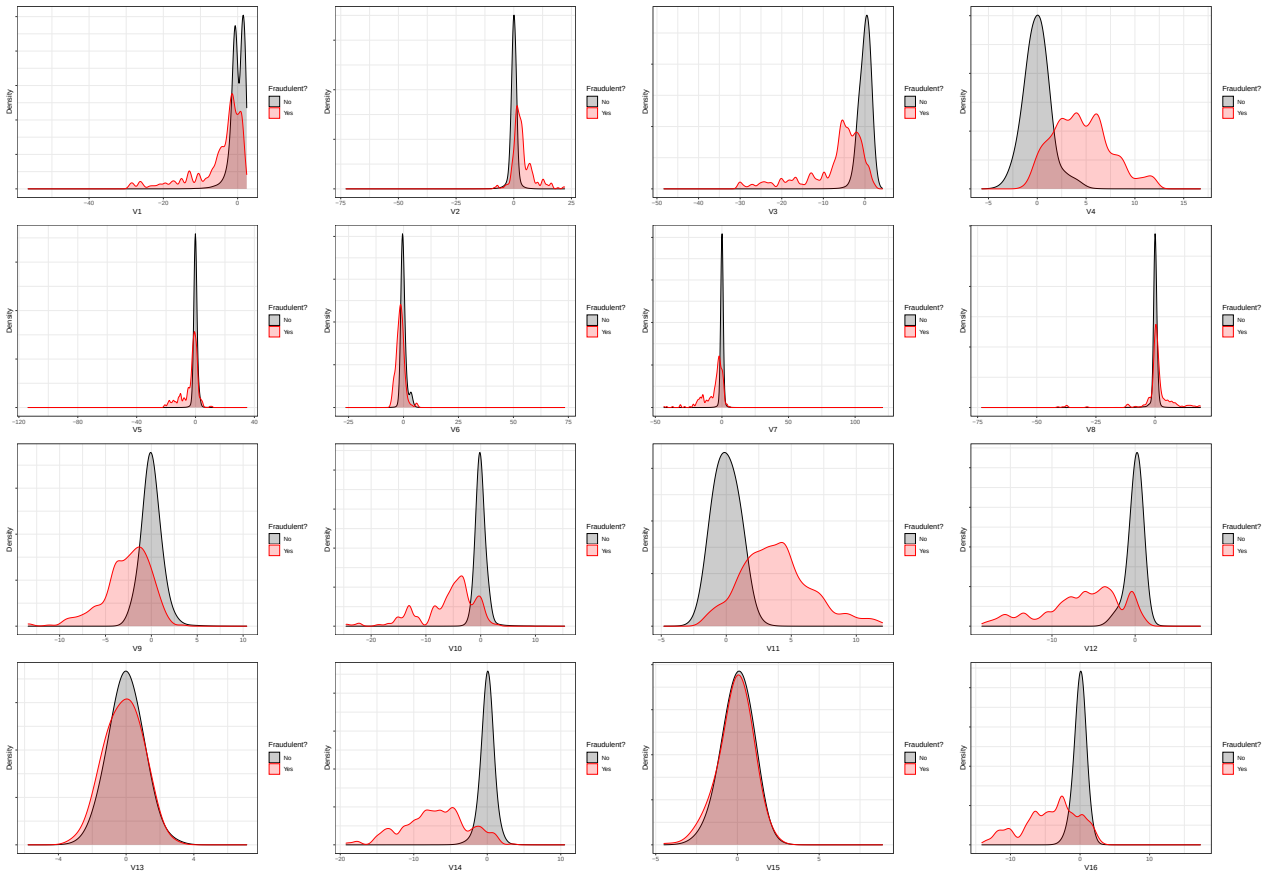
With the newly created training set we will take a closer look at the distribution of fraudulent cases in the different principal components. Therefore, we create a function that plots a scatterplot of PC_i vs PC_{i+1} with color denoting if the transaction was fraudulent. A second funtion will look at distribution estimates of the different principal components, color again denoting if the transaction was fraudulent

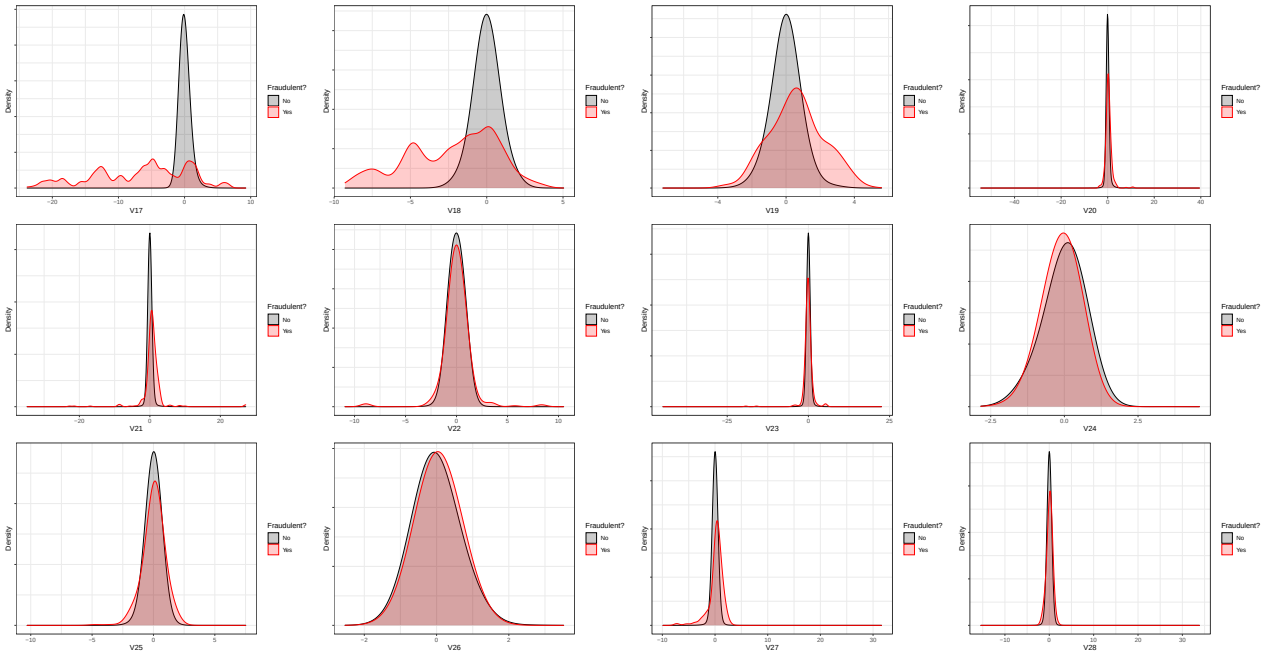
```
# depending on the performance of your PC this might take some time!
do_plot_scatter <- function(i) {
  data_train %>% ggplot(aes(x = data_train[,i], y = data_train[,i+1])) +
    geom_point(aes(color = Class), alpha = 0.1) +
    geom_point(aes(color = Class), alpha = ifelse(data_train$Class == "Fraud", 0.5, 0)) +
    xlab(paste0(colnames(data_train)[i])) +
    ylab(paste0(colnames(data_train)[i+1])) +
    theme_bw() +
    scale_color_manual(name = "Fraudulent?", values = c("black", "red"),
                       labels = c("Legitimate" = "No", "Fraud" = "Yes"))
```

```
}
sapply(seq(2, 28, by = 2), function(x) {plot(do_plot_scatter(x))})
```



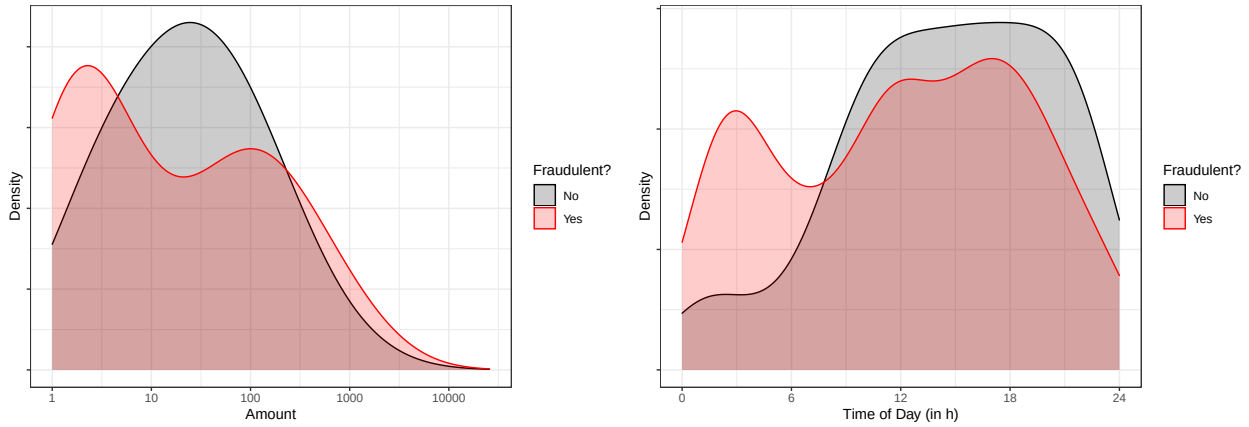
```
do_plot_distribution <- function(i) {
  limits0 <- data_train[data_train$Class == 0, i] %>% quantile(c(0.25, 0.975))
  limits1 <- data_train[data_train$Class == 1, i] %>% quantile(c(0.25, 0.975))
  limits <- c(min(c(limits0, limits1)), max(c(limits0, limits1)))
  data_train %>% ggplot(aes(x = data_train[,i], color = Class, fill = Class)) +
    geom_density(bw = 0.5, alpha = 0.2) +
    scale_x_continuous(paste0(colnames(data_train)[i])) +
    scale_y_continuous("Density", labels = NULL) +
    theme_bw() +
    scale_color_manual(name = "Fraudulent?", values = c("black", "red"),
                       labels = c("Legitimate" = "No", "Fraud" = "Yes")) +
    scale_fill_manual(name = "Fraudulent?", values = c("black", "red"),
                      labels = c("Legitimate" = "No", "Fraud" = "Yes")) +
    coord_cartesian(xlim = limits)
}
sapply(2:(ncol(data_train)-2), function(x) {plot(do_plot_distribution(x))})
```





We notice two things: first, some principal components seem to have highly different distributions for fraudulent and non-fraudulent cases. Second, some features don't seem to follow a normal distribution, let alone multivariate normality. Therefore, algorithms that don't expect normality (kNN, decision trees) may perform better than those that do (LDA, logistic regression). However, the deviance in the tails of the distribution is to a large extend due to Fraud. Therefore, the impact of non-gaussian distributions has to be explored by applying the different algorithms.

Upon inspecting `Amount` and `Time`, we see that cases of Fraud appear to take place in the night and have lower value than non-fraudulent transactions:



Prediction Methods

Baseline Measurements

The most simple prediction one could make is always guessing the *a priori*, the Class with the highest propability:

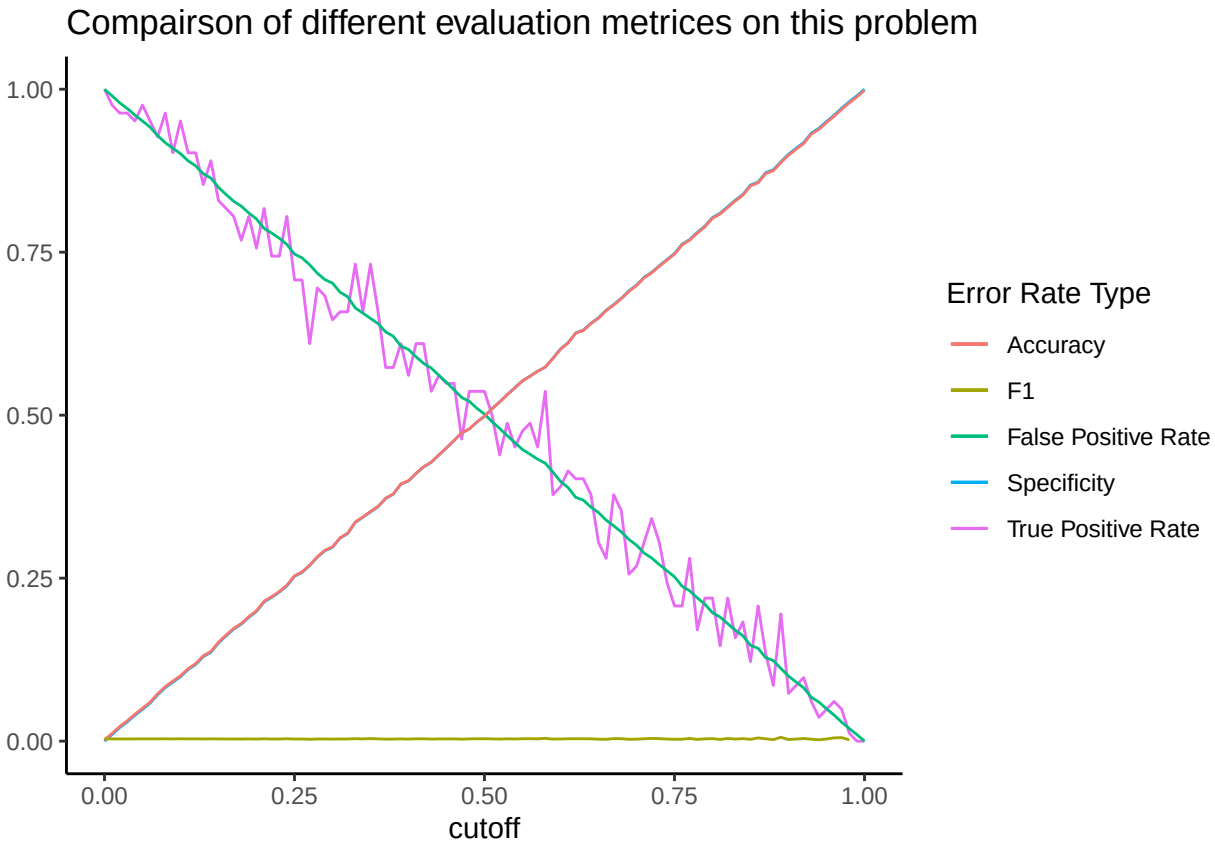
```
predictions <- data.frame(guessing = sample(unique(data_train[,31]), prob = c(1, 0),
                                            size =  nrow(data_test), replace = TRUE))
caret::confusionMatrix(predictions$guessing, data_test$Class, positive = "Fraud")
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   Legitimate Fraud
##   Legitimate      47386    82
##   Fraud               0     0
##
##                Accuracy : 0.9983
##                  95% CI : (0.9979, 0.9986)
##     No Information Rate : 0.9983
##     P-Value [Acc > NIR] : 0.5293
##
##                   Kappa : 0
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.000000
##             Specificity : 1.000000
##          Pos Pred Value :      NaN
##          Neg Pred Value : 0.998273
##              Prevalence : 0.001727
##          Detection Rate : 0.000000
##    Detection Prevalence : 0.000000
##       Balanced Accuracy : 0.500000
##
##        'Positive' Class : Fraud
##
```

We see that we can achieve an accuracy of 99.8% without any information on the transaction. However this comes at the price of not detecting and preventing any fraudulent transactions. We could also guess with different propabilites p for "Fraud" and "Legitimate" and see how different metrics of prediction performance behave:

```
suppressWarnings(set.seed(42, sample.kind = "Rounding"))
ps <- seq(0,1, by = 0.01)
guessing <- map_df(ps, function(p){
  y_hat <- sample(unique(data_train[,31]), prob = c(p, 1-p),
                  size =  nrow(data_test), replace = TRUE)
```

```
  SS <- caret::confusionMatrix(y_hat, data_test$Class,
                               positive = "Fraud")$byClass[
                                 c("Sensitivity", "Specificity", "F1")]
  AC <- caret::confusionMatrix(y_hat, data_test$Class,
                               positive = "Fraud")$overall["Accuracy"]
  list(method = "guessing", cutoff = p, sensitivity = SS[1],
       specificity = SS[2], F1 = SS[3], accuracy = AC[1])
})
guessing %>% ggplot(aes(x = cutoff)) +
  geom_line(aes(y = sensitivity, color = "True Positive Rate")) +
  geom_line(aes(y = 1 - specificity, color = "False Positive Rate")) +
  geom_line(aes(y = specificity, color = "Specificity")) +
  geom_line(aes(y = accuracy, color = "Accuracy")) +
  geom_line(aes(y = F1, color = "F1")) +
  theme_classic() + ylab(NULL) +
  scale_color_discrete("Error Rate Type") +
  ggtitle("Compairson of different evaluation metrices on this problem")
```

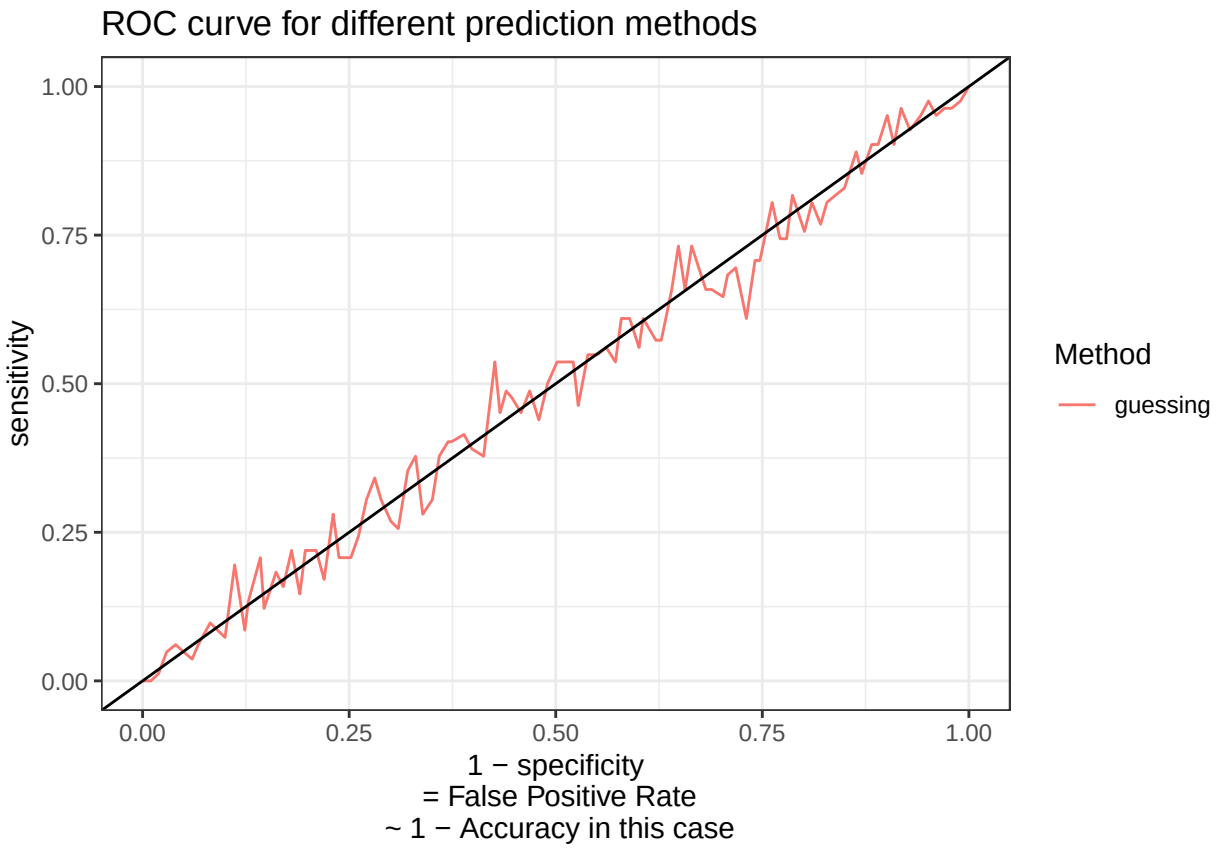


We see that accuracy and specificity (= 1 - true positive rate) are almost identical. This is due to the large bias in prevalences, with legitimate transactions dominating the accuracy. In this scenario the use of a ROC, where sensitivity is plotted against 1 - specificity, seems to be the way to go to assess prediction performance.

As we will be using the ROC throughout this report, we will create a function that automatically draws a

ROC for us and apply it to the guessing approach:

```
# define a function that draws a ROC curve
do_ROC <- function(dataframe){
  dataframe %>% ggplot(aes(x = 1 - specificity, y = sensitivity, color = method)) +
  geom_line() +
  geom_abline(intercept = 0, slope = 1) +
  theme_bw() +
  scale_color_discrete("Method") +
  xlab("1 - specificity\n = False Positive Rate\n ~ 1 - Accuracy in this case") +
  ggtitle("ROC curve for different prediction methods")
}
# plot the guessing approach
do_ROC(guessing)
```



We see that the *guessing* method "wiggles" around the identity line (or no-information-line). This "wiggelieness" or noise is due to the very small test set of Fraud cases. Although the test set contains 47468 observations, fewer than a hundred of them are fraudulent. Therefore, the estimate for sensitivity contains quite a lot of noise. Keep this in mind as we will progress along different ML algorithms and compaire their performance.

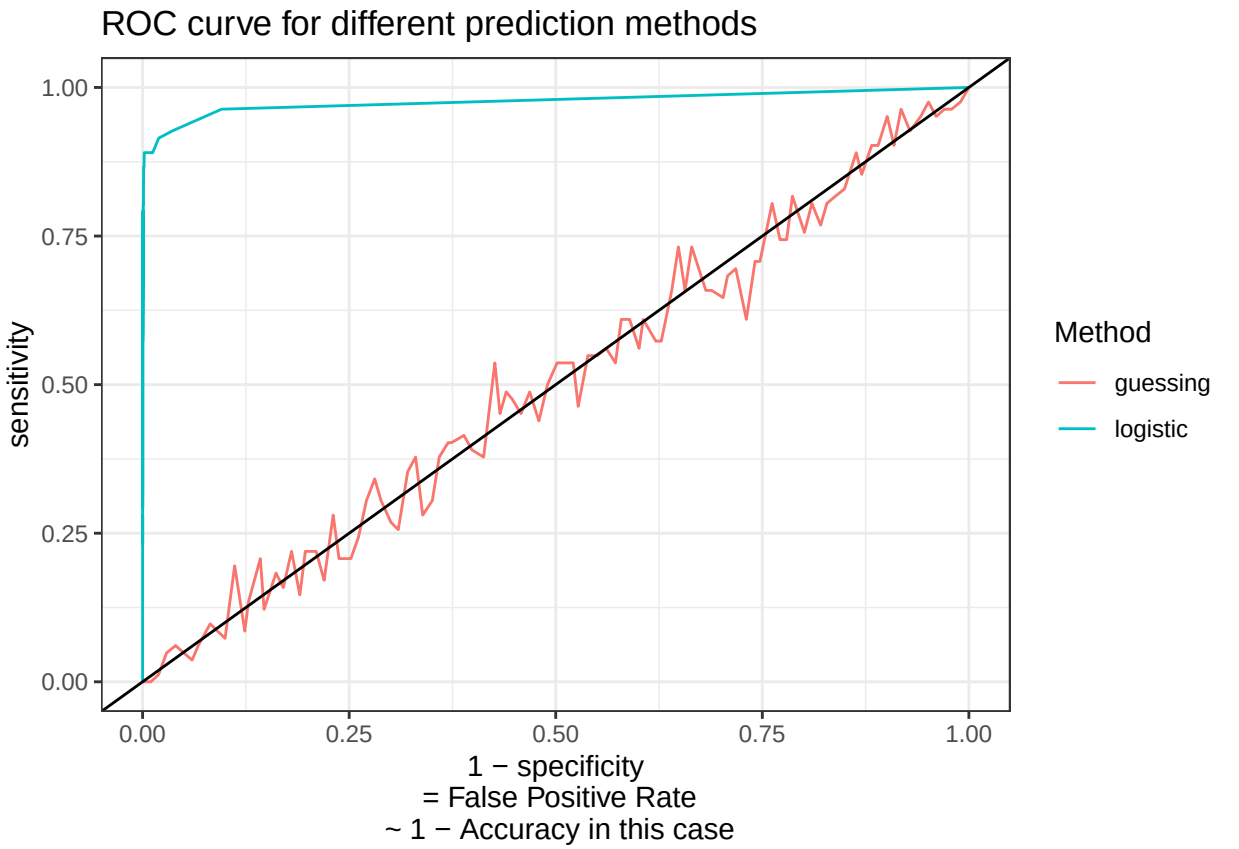
The most simple machine learning algorithm is logistic regression. By adding up the predictors times the estimated coefficients, one can quite easily create a probability for a given class. Therefore, this algorithm is highly interpretable and will serve as a benchmark. If more complex prediction methods cannot beat

logistic regression, we will stick with it due to its high interpretability. We will, again, define a function that evaluates the performance of a probability estimate as we will use this throughout the whole report:

```
# We will write a function that automatically evaluates the algorithm's
# probability esimations with different cutoff values
do_eval <- function(data, method_name = "please specify", ps = seq(0,1, by = 0.001)){
  output <- map_df(ps, function(p){
    y_hat <- ifelse(data < p, "Legitimate", "Fraud") %>%
      factor(levels = levels(data_train$Class))
    SS <- caret::confusionMatrix(y_hat, data_test$Class,
                                 positive = "Fraud")$byClass[
                                   c("Sensitivity", "Specificity", "F1")]
    AC <- caret::confusionMatrix(y_hat, data_test$Class,
                                 positive = "Fraud")$overall["Accuracy"]
    list(method = method_name, cutoff = p, sensitivity = SS[1],
         specificity = SS[2], F1 = SS[3], accuracy = AC[1])
  })
  return(output)
}
```

When applying logistic regression, we see an huge increase of performance over the random guessing approach:

```
suppressWarnings(set.seed(42, sample.kind = "Rounding"))
fit_logistic <- glm(Class ~ ., data = data_train, family = "binomial")
y_hat_p <- predict(fit_logistic, newdata = data_test, type = "response")
ROC_all <- rbind(guessing, do_eval(data = y_hat_p, method_name = "logistic"))
do_ROC(ROC_all)
```



Remember that the goal is to have perfect sensitivity for all values of specificity. This means a line that shoots straight into the upper left corner and stays at a sensitivity of 1 would be ideal. The logistic regression seems to be able to capture about 80% of the fraudulent transactions without generating large amounts of false positive cases. The last 20%, however, seem not detectable with ordinary logistic regression.

Advanced Machine Learning Techniques 1: Linear Model Selection and Motivation for Sampling Techniques

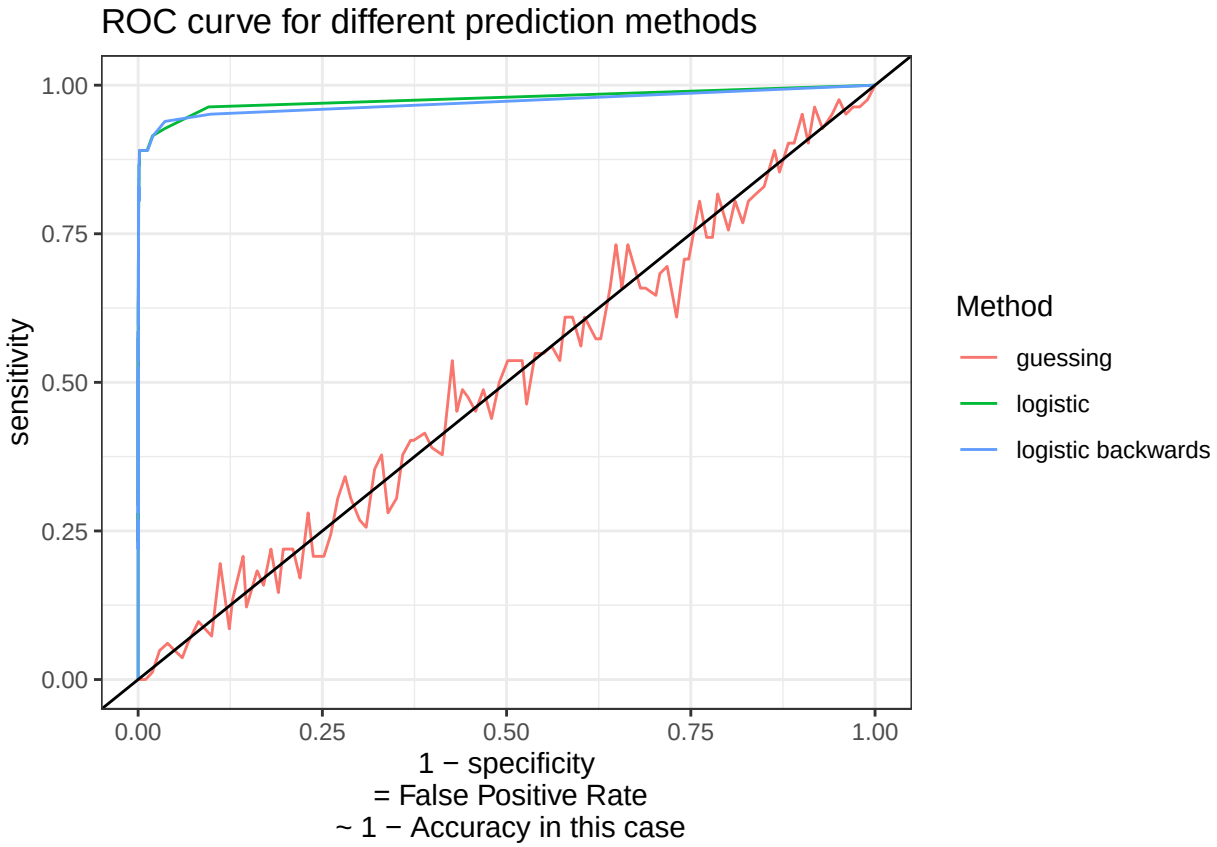
We utilized 30 features for our logistic regression. There is a decent chance that we overfit our model, meaning we captured trends and relationships that were only generated by chance, by random noise. In this part we will discuss different possibilities to deal with overfitting linear models. This will also serve as demonstration for why sampling techniques can be a crucial part of data analysis when dealing with highly unbalanced classes.

The goodness of fit (RMSE for ordinary and deviance for logistic regression) must increase as we feed the model additional features. However, the goodness of fit will also increase when the additional features are completly random. To account for this random increase of fit, different measures have been developed such as the AIC, BIC, or adjusted R^2[6]. For demonstration purposes, we will use the AIC. Feature selection tries to optimize the linear regression (= optimize the AIC) by removing predictors and only using those that are needed. One could try to fit every possible model, which would lead to $31! \approx 8 * 10^{33}$ models. As this is computationally not feasible, we will start with the full model (all n features) and try leaving out different

[6]Explaining all the theory behind the used techniques is beyond the scope of this report. Please refer to *Elements of Statistical Learning* (James et.al, 2017), which is openly available and provides a good summary of the widely used algorithms

predictors, one at a time. The best model is then taken, and we iterate through this process until we find the optimum of the AIC. With this *backward selection* will therefore only fit a maximum of $\sum 30, 29, ..., 1 = 465$ models, making the computing time much faster:

```
suppressWarnings(set.seed(42, sample.kind = "Rounding"))
logistic_backwards <- MASS::stepAIC(fit_logistic, direction = "backward")
y_hat_p <- predict(fit_logistic, newdata = data_test, type = "response")
ROC_all <- rbind(ROC_all, do_eval(data = y_hat_p, method_name = "logistic backwards"))
do_ROC(ROC_all)
```

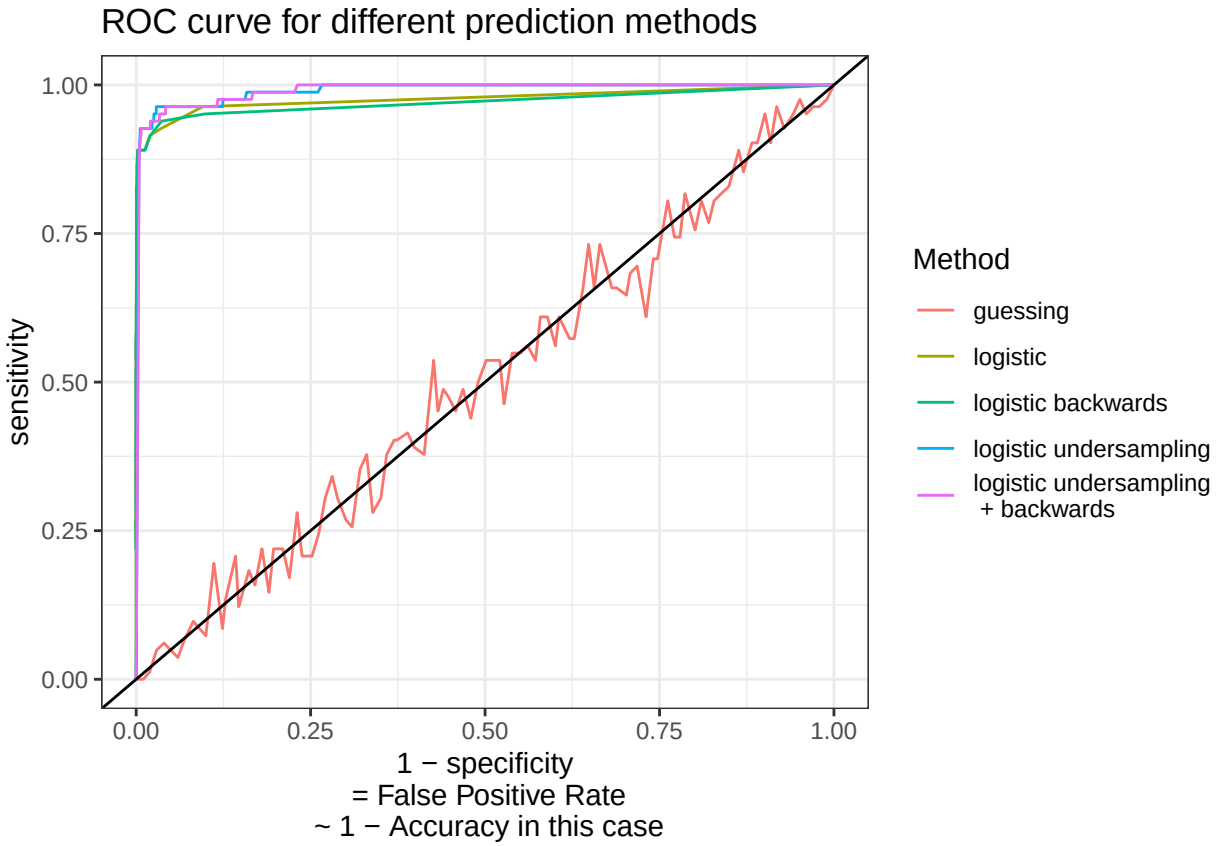


However, this approach still takes time as the training data consists of 189871 observations, of which 328 are of high interest - the cases of fraud. When facing highly unbalanced datasets sampling techniques are often applied. One could generate artificial cases of fraud (*oversampling*) or select only a subset of the legitimate transactions (*undersamling*). As we want to decrease processing time, the second option can be applied to our dataset in order to create a small dataset with a fraud prevalence of about 33%:

```
suppressWarnings(set.seed(42, sample.kind = "Rounding"))
data_train_under <- ROSE::ovun.sample(Class ~ ., data_train, method = "under", p = 1/3)$data
```

When applying both logistic regressions again we can see that loosing those 95% of our data does not harm our prediction but actually increases the prediction performance:

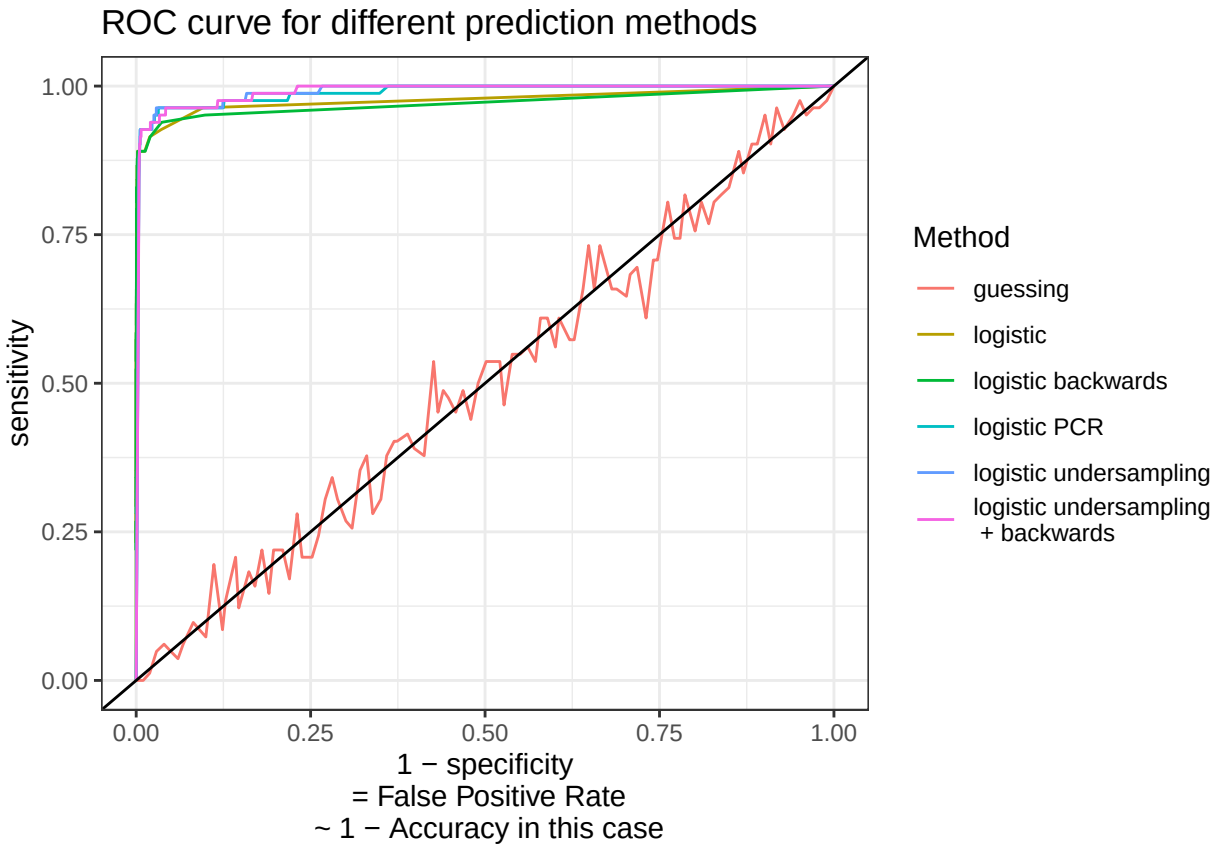
```r
# plain vanilla logistic regression
suppressWarnings(set.seed(42, sample.kind = "Rounding"))
fit_logistic <- glm(Class ~ ., data = data_train_under, family = "binomial")
y_hat_p <- predict(fit_logistic, newdata = data_test, type = "response")
ROC_all <- rbind(ROC_all, do_eval(data = y_hat_p, method_name = "logistic undersampling"))

# backwards selection logistic regression
fit_logistic <- MASS::stepAIC(fit_logistic, direction = "backward")
y_hat_p <- predict(fit_logistic, newdata = data_test, type = "response")
ROC_all <- rbind(ROC_all, do_eval(data = y_hat_p,
                                  method_name = "logistic undersampling\n + backwards"))
do_ROC(ROC_all)
```



Advanced Machine Learning Techniques 2: PC-Regression, Ridge and Lasso

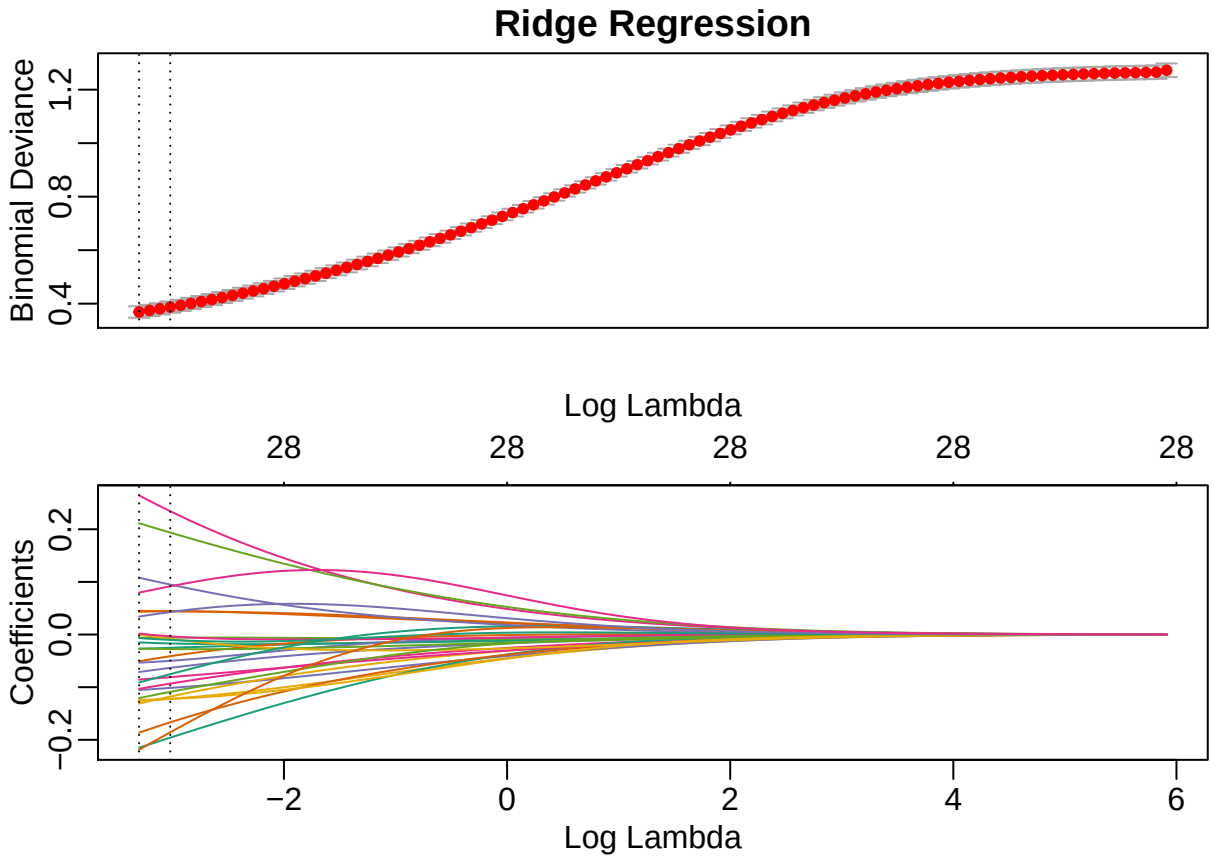
Instead of deleting whole predictors one could also use a principal component regression. Here, one computes all n Principal Components and use only the first $m < n$ PCs. One advantage over feature selection is that the number of possible models decreases to 30. As the PCs are ordered, only n models have to be fitted. As we already deal with principal components, we can easily write our own principal component regression, utilizing the AIC as a model performance metric:

```r
# Baseline Model contains Time and Amount
X <- data_train_under %>% select(Time, Amount, Class)
X_holdout <- data_train_under %>% select(-Time, -Amount, -Class)
PC_eval <- map_df(1:ncol(X_holdout), function(i){
  # in each step 1 PC is added on top of the already existing ones
  X_tmp <- cbind(X, X_holdout %>% select(1:i))
  fit_tmp <- glm(Class ~ . , data = X_tmp, family = "binomial")
  list(call = paste0(fit_tmp$terms)[3], AIC = fit_tmp$aic)
})
best_call <- PC_eval %>% top_n(1, -AIC) %>% pull(call)

# use the best call and fit this model
suppressWarnings(set.seed(42, sample.kind = "Rounding"))
fit_logistic <- glm(as.formula(paste0("Class ~ ",best_call)),
                    data = data_train_under, family = "binomial")

y_hat_p <- predict(fit_logistic, newdata = data_test, type = "response")
ROC_all <- rbind(ROC_all, do_eval(data = y_hat_p, method_name = "logistic PCR"))
do_ROC(ROC_all)
```

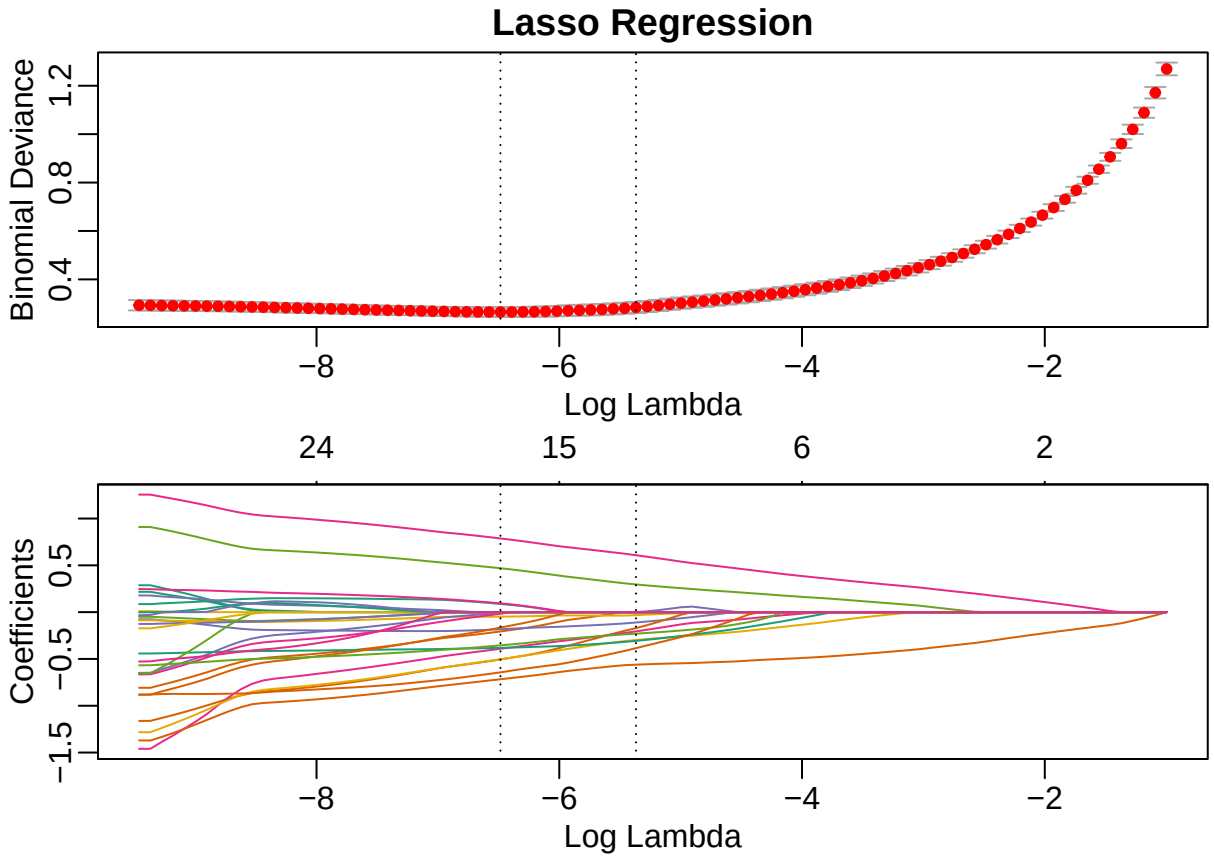


In this case the principal components regression is a special case of feature selection, as the original features were already the PCs. However, in most applications PC regression can be though of as "shinking" the

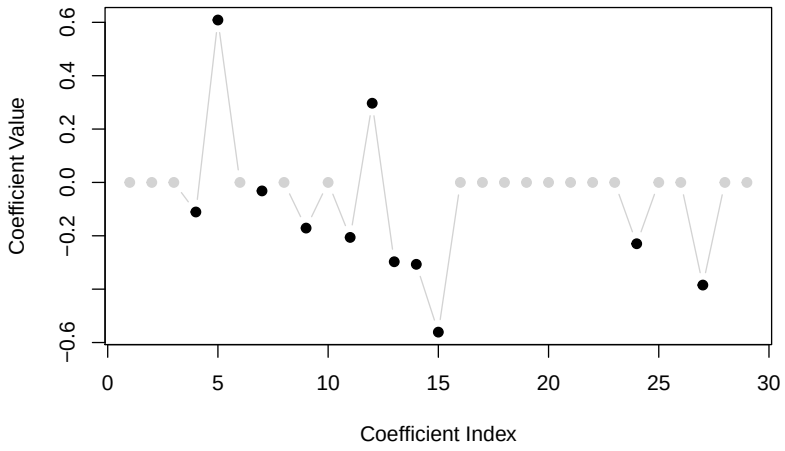
coefficients of some original features[7]. Other common *shrinkage* or *regularization* methods include Ridge (L2 Norm shrinkage) and Lasso (L1 Norm Shrinkage) regression. With b_i being the coefficient for feature i, the Ridge tries to minimize $RSS + \lambda \sum b_i^2$ whilst the Lasso minimizes $RSS + \lambda \sum |b_i|$. The difference may be subtle but Ridge-Regression produces more non-zero coefficients. Which of both performs better and which lambda is optimal can assessed by cross validation:



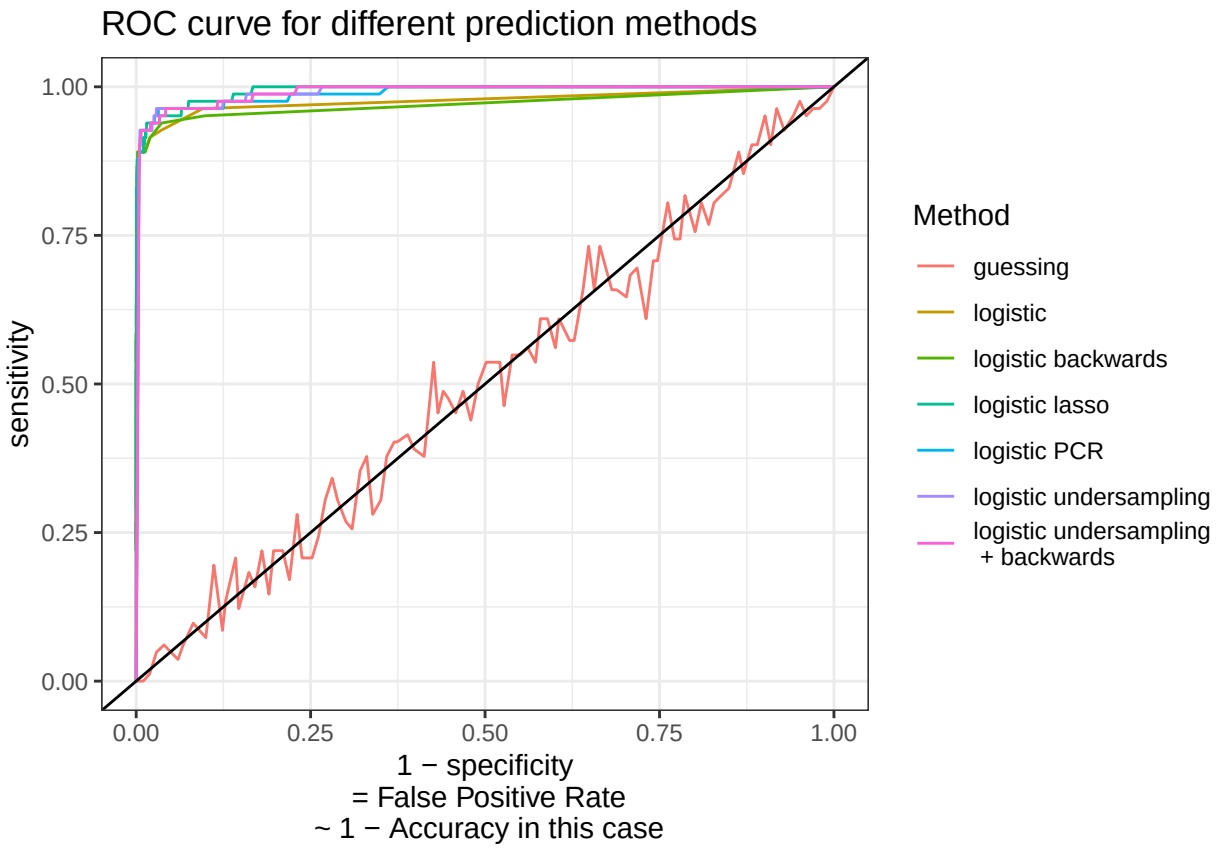
[7]For further information on this please consult an in-depth book



The first vertical line indicates the best performing lambda, the second line indicates 1 standard error more conservative than the best performing lambda. Whilst Ridge does not seem to improve the performance at all, Lasso may increase the performance a little bit with an optimal lambda of about 0.002. It did, however, also set some coefficients to zero:



And again, the resulting model with fewer predictors still has a very good performance:



Advanced Machine Learning Techniques 3: Beyond linearity

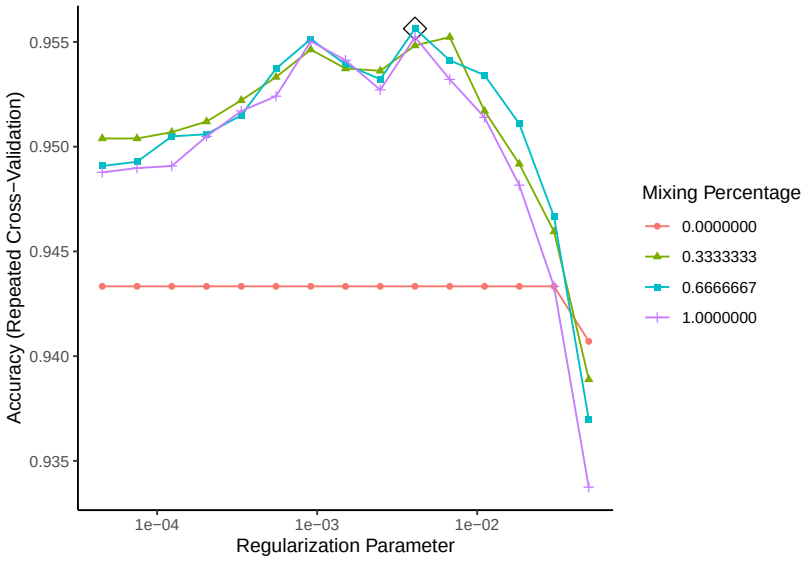
Linear models have the advantage of being highly interpretable. On the other hand are machine learning algorithms that work miraculous - they yield good performance but you won't know how. They are not constrained by the prerequisites of classic regression analysis such as linear boundaries. We will compaire logistic regression to k-nearest-neighbours, random forests and regression trees. Therefore, we will use the `caret` package which allows for an coherent syntax. As our primary goal is a compairson and not very fine parameter tuning, we will only use 10-fold cross validation and a low-resoluting tune grid.

```
# We will use some fast 10-fold CV to get first impressions of the algorithm performance
control_parameters <- caret::trainControl(method = "repeatedcv", number = 10, repeats = 10)

# We will set up some tuning parameter grids
grid_plainvanilla_reg <- expand.grid(alpha = 0, lambda = 0) # this equals regular regression
grid_glmnet <- expand.grid(alpha = seq(0, 1, by = 1/3),
                           lambda = exp(seq(-10, -3, by = 0.5)))
grid_knn <- expand.grid(k = seq(3, 51, by = 2))
grid_GBM <-
  expand.grid(n.trees = c(5000),        # those values were selected directly with the
              interaction.depth = c(5), # gbm package as this takes very long with caret
```

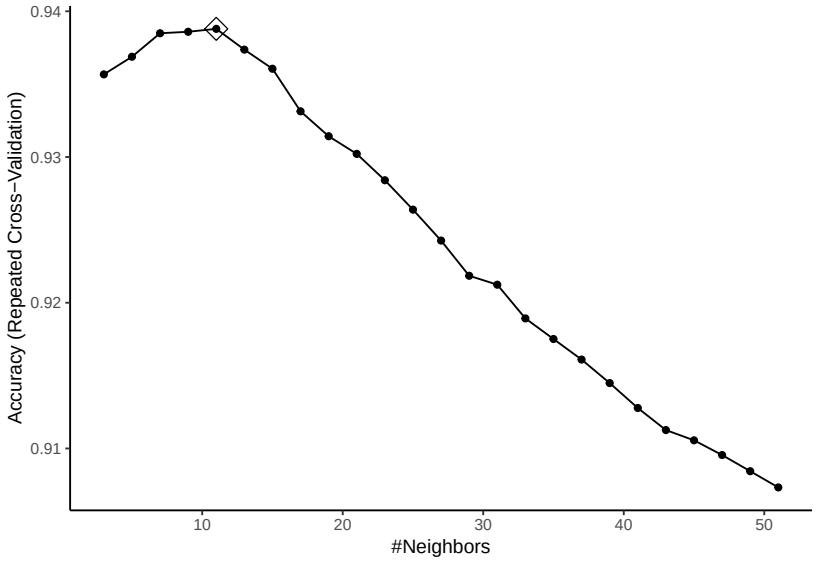
```
              shrinkage = c(0.001),     # You don't have to cross-validate as you can
              n.minobsinnode = c(17))   # use the out of bag esimate for the performance
grid_RF <- expand.grid(mtry = c(1, 3, 5, 7))
```

With the tuning parameters set, we can apply various different machine learning algorithms with very little effort and compaire their results:

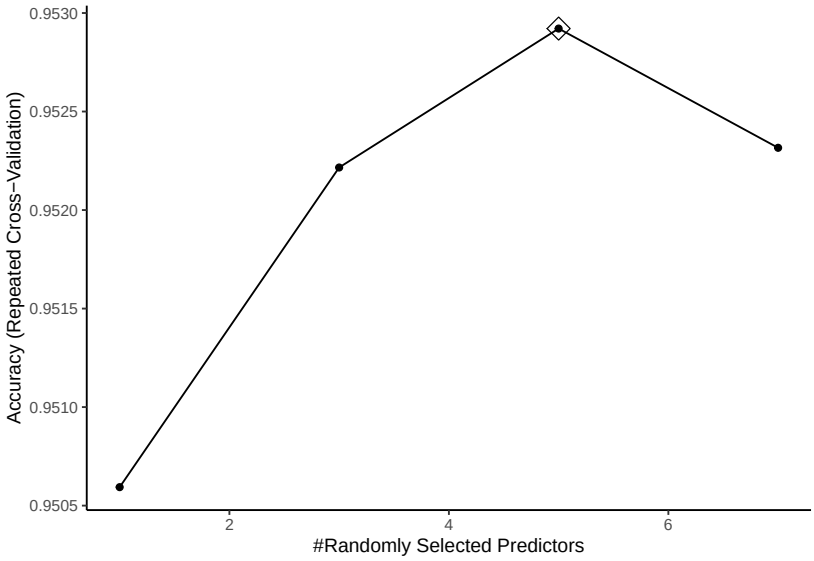
```
# Do Plain Vanilla Logistic Regression with caret package
suppressWarnings(set.seed(999, sample.kind = "Rounding"))
fit_plainvanilla_reg <- caret::train(Class ~., data = data_train_under,
                                     method = "glmnet", tuneGrid = grid_plainvanilla_reg,
                                     preProcess = c("scale", "center"),
                                     trControl = control_parameters)
p_Fraud <- predict(fit_plainvanilla_reg, newdata = data_test, type = "prob")[,2]
ROC_caret <- do_eval(data = p_Fraud, method_name = "Plain Vanilla\nLogistic Regression")


# Do Elastic Net Logistic Regression with caret package
suppressWarnings(set.seed(999, sample.kind = "Rounding"))
fit_glmnet <- caret::train(Class ~., data = data_train_under,
                           method = "glmnet", tuneGrid = grid_glmnet,
                           preProcess = c("scale", "center"),
                           trControl = control_parameters)
p_Fraud <- predict(fit_glmnet, newdata = data_test, type = "prob")[,2]
ROC_caret <- rbind(ROC_caret, do_eval(data = p_Fraud,
                                      method_name = "Elastic Net\n Regression"))

ggplot(fit_glmnet, highlight = TRUE) + theme_classic() + scale_x_continuous(trans = "log10")
```



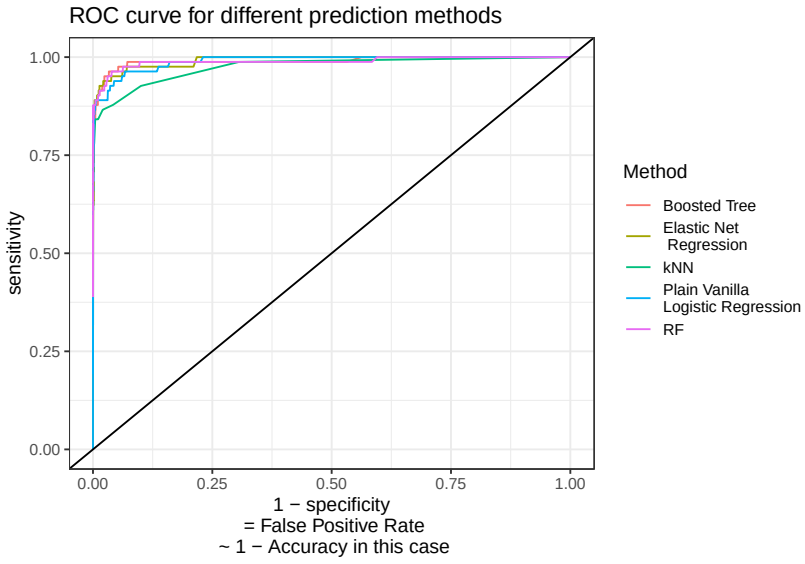
```r
# Do kNN with caret package
suppressWarnings(set.seed(999, sample.kind = "Rounding"))
fit_knn <- caret::train(Class ~., data = data_train_under,
                        method = "knn", tuneGrid = grid_knn,
                        preProcess = c("scale", "center"), trControl = control_parameters)
p_Fraud <- predict(fit_knn, newdata = data_test, type = "prob")[,2]
ROC_caret <- rbind(ROC_caret, do_eval(data = p_Fraud, method_name = "kNN"))

ggplot(fit_knn, highlight = TRUE) + theme_classic()
```




```r
# Do Random Forest with caret package
suppressWarnings(set.seed(999, sample.kind = "Rounding"))
fit_RF <- caret::train(Class ~., data = data_train_under,
                        method = "rf", tuneGrid = grid_RF, ntree = 1000,
                        preProcess = c("scale", "center"), trControl = control_parameters)
p_Fraud <- predict(fit_RF, newdata = data_test, type = "prob")[,2]
ROC_caret <- rbind(ROC_caret, do_eval(data = p_Fraud, method_name = "RF"))

ggplot(fit_RF, highlight = TRUE) + theme_classic()
```

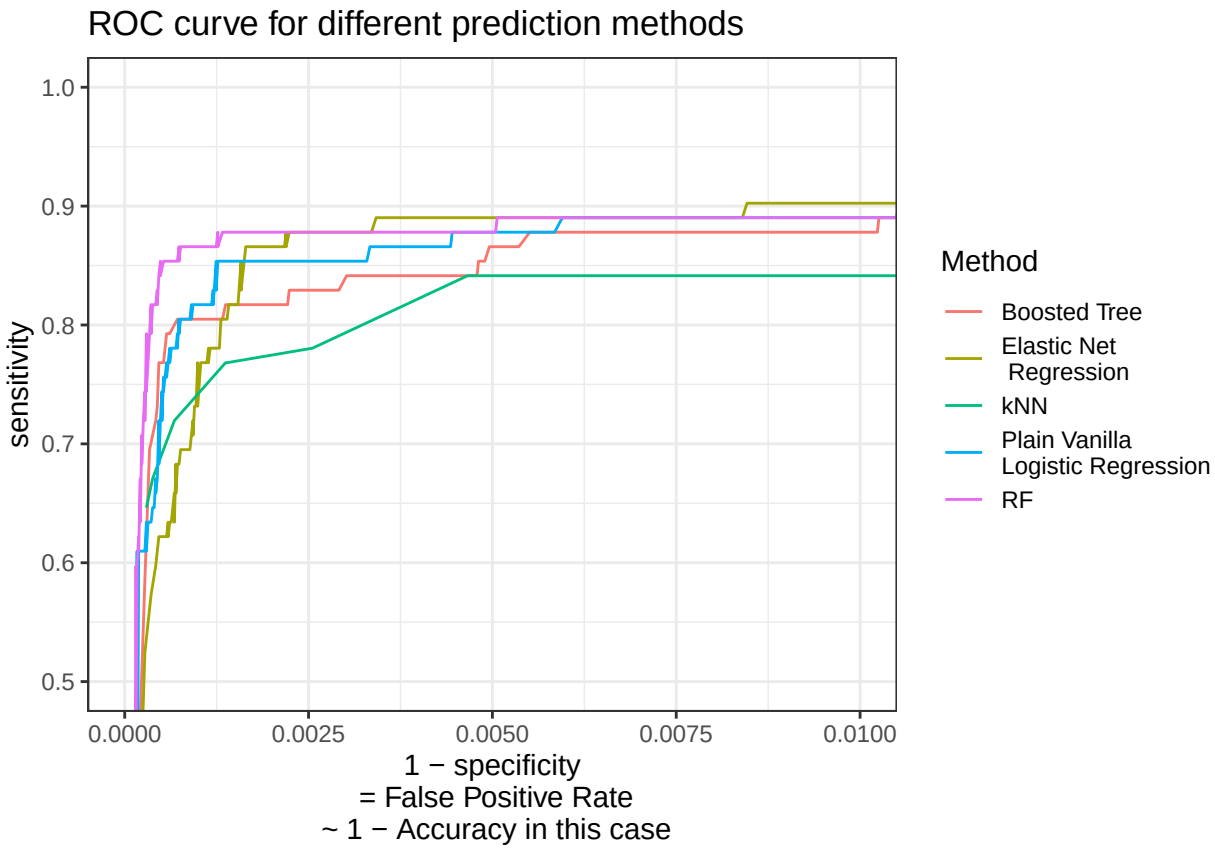



```
# Do Boosting with caret package
suppressWarnings(set.seed(999, sample.kind = "Rounding"))
fit_gbm <- caret::train(Class ~., data = data_train_under,
                        method = "gbm", verbose = FALSE, distribution = "bernoulli",
                        tuneGrid = grid_GBM,
                        preProcess = c("scale", "center"),
                        trControl = control_parameters)
p_Fraud <- predict(fit_gbm, newdata = data_test, type = "prob")[,2]
ROC_caret <- rbind(ROC_caret, do_eval(data = p_Fraud, method_name = "Boosted Tree"))


# Plot the results
do_ROC(ROC_caret)
```



We will focus on the performance for specificities > 0.99, as one would not want too many false-positive calls:

```
do_ROC(ROC_caret) + coord_cartesian(xlim = c(0, 0.01), ylim = c(0.5, 1))
```



Discussion and Limitations

Logistic regression served as a baseline model, as it is highly interpretable. It performed quite good for up to 80% sensitivity but then the performance did not further improve. Lasso regression did not alter the performance much but made the model even more interpretable by creating a sparse coefficient matrix, meaning only fewer features had an influence on the prediction. The kNN-algorithm does not seem applicable to this particular problem as it performs worse for almost all cutoff values. One could try kNN with fewer features (possibly the ones from the Lasso regression) and see if the performance increases as it softens the course of dimensionality. Boosted Classification Trees perform about the same as logistic regression. Only random forests appear to surpass the performance of logistic regression. Why classification trees do not benefit from boosting in this particular case is not completely understandable for me. One reason might be: the observable difference is only due to chance. Another explaination is that we overfit the data. Unlike random forest, boosting builds upon the residuals from previously generated models and can therefore be overfit. Maybe accuracy, which we used as a measure in cross-validation, was not optimal for tuning.

In my opinion, the best algorithms for this particular problem are the Lasso and random forests. Whilst random forests provide the best prediction, the Lasso manages to reduce prediction to only a handful of relevant features. Thus, when explaining why a particular transaction was flagged as "Fraud", the Lasso can provide a answer. As we do not have the original features but only the principal components that stem from them, this advantage of the Lasso is lost for this dataset. Also, for random forest, an inspection of the feature importance will provide no further insights. As interpretability is not achievable anyways, random forest is chosen to be the method that is applied onto the validation set.

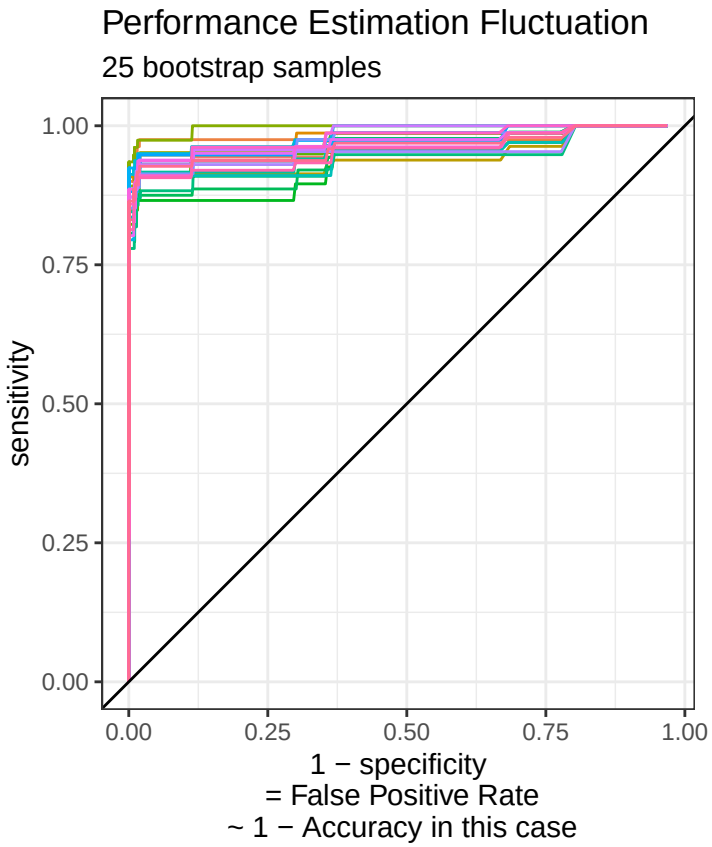
Two questions remain:

1. How does the algorithm perform on new data and how much does its performance estimation fluctuate?
2. Which cutoff-values does one decide on in real-world applications?

Whilst we can answer the first question via bootstrapping on the validation set,

```
suppressWarnings(set.seed(10101, sample.kind = "Rounding"))
# 25 bootstrap samples
for (i in c(1:25)){
  # create bootstrap sample
  BT_index <- sample(c(1:nrow(data_validation)), nrow(data_validation), replace = TRUE)
  BT_data <- data_validation[BT_index,]
  # predict outcome for this sample with the RF created above
  p_Fraud <- predict(fit_RF, newdata = BT_data, type = "prob")[,2]
  # do evaluation of prediction
  ps = seq(0,1, by = 0.001)
  tmp <- map_df(ps, function(p){
    y_hat <- ifelse(p_Fraud <= p, "Legitimate", "Fraud") %>%
      factor(levels = levels(data_train$Class))
    SS <- caret::confusionMatrix(y_hat, BT_data$Class,
                                 positive = "Fraud")$byClass[
                                   c("Sensitivity", "Specificity", "F1")]
    AC <- caret::confusionMatrix(y_hat, BT_data$Class,
                                 positive = "Fraud")$overall["Accuracy"]
    list(method = paste0("Bootstrap Sample ", i), cutoff = p, sensitivity = SS[1],
         specificity = SS[2], F1 = SS[3], accuracy = AC[1])
```

```
  })
  # save the evaluation
  if (i == 1){
    BT_results <- tmp
  }
  if (i != 1){
    BT_results <- rbind(BT_results, tmp)
    tmp <- NULL
  }
}


# plot the results
do_ROC(BT_results) + theme(legend.position = "none") + coord_equal() +
  ggtitle("Performance Estimation Fluctuation", subtitle = "25 bootstrap samples")
```



the second question is a question of weighting up the costs of false-positive versus false-negative cases. This remains to be decided by the financial institution incorporating the fraud detection.

Conclusion

Our algorithm managed to flag quite many fraudulent transactions as Fraud. However, when wanting to detect more than 80% of the fraudulent transactions. this comes at the price of lower specifictiy and thus many mistakenly frozen accounts. In reality, cutoff values are chosen such that even the most state of the art machine learning techniques only detect 2 out of every 3 dollars of attempted fraud[8]. Therefore, be aware of the consumer information the FBI issues[9]. On the other hand, also notify your bank when your transaction behaviour is about to change. Their algorithms do not know whether you are on a exotic vacation or a fraudster is scamming you out of money on the cayman islands. Although this report does not reach the performance of currently employed algorithms, it may help you gain an understanding of the algorithms and methods used to prevent fraud. This insight is going to help you minimize the risk of false-negative and false-positive errors. And for you this directly related to less lost money and fewer inconvenient situations on holiday.

[8] https://www.ukfinance.org.uk/system/files/Fraud%20The%20Facts%202019%20-%20FINAL%20ONLINE.pdf
[9] https://www.fbi.gov/scams-and-safety/common-scams-and-crimes/credit-card-fraud