Humboldt University Berlin

Term paper

Credit Default Prediction

Valeryia Mosinzova, Michail Psarakis, Thomas Siskos

Statistical Programming Languages

supervised by
Alla Petukhina

March 18, 2018

Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

The term default probability is a financial term which describes the likelihood that a borrower will fail to make scheduled repayments over a specific time horizon, which is usually a year. The effective estimation and prediction of corporate defaults are crucial for asset pricing, credit risk assessment of loan portfolios as well as the valuation of other financial products exposed to corporate defaults (Miao et al., 2018). This issue has been considered in several studies. There are two basic approaches to deal with default risk analysis: the market-based model (a.k.a. structural model) and the statistical approach, determined through the empirical analysis of historical data like accounting data (Härdle et al., 2012).

In our paper we aim to perform default prediction analysis by following the methodologies of Härdle et al. (2012), Chen et al. (2011) and Zhang & Härdle (2010) We seek to replicate some of their results using the `R` programming language. Our analysis is based on the creditreform-database, which provides several financial statement variables of German firms. In our work we perform linear and generalized linear techniques for predicting bankruptcy based on financial ratios. We compare the predictive performance between following models: logit, probit, Classification and Regression Trees (CART), Random Forests (RF) and Linear Discriminant Analysis (LDA).

The rest of the paper is organized as follows. In the next section we describe the creditreform-dataset. In section 3 we explain the data preparation procedures as well as the variables and ratios used in this paper. Section 4 contains a brief introduction to Binary Response Models, with a custom implementation and an application of a Logit model to the creditreform-dataset. In section 5 we apply Classification and Regression Trees (CART), which is followed by a section about Random Forests (RF). Linear Discriminant Analysis (LDA) is briefly explained and applied in section 6. In section 7, we present the steps we take to evaluate predictions of the selected models and the suitable performance measures. Finally section 8 summarises the conclusions we draw from our analysis.

2 The dataset

We used the creditreform database obtained from the Laboratory for Empirical and Quantitative Research (LEQR) of the Humboldt University of Berlin (leqr.wiwi.hu-berlin.de) to analyse a sample of 20,000 solvent and 1,000 insolvent German firms from the period of 1996-2007. Due to incomplete data from 2003 onwards and missing data for insolvent firms in 1996 we will focus our analysis on the data of the period between 1997 and 2002. About half of the data refers to the years 2001 and 2002. The majority of firms appear several times in different years in the dataset, while the data for the insolvent firms was collected two years prior to the default (Chen et al. (2011)). Each firm is described by several financial statement variables as those in balance sheets and income statements. A complete list of all variables of the creditreform database as well as their descriptions is provided in table 1.

The firms are divided into the sectors of construction (39.7%), manufacturing (25.7%), wholesale and retail trade (20.1%), real estate (9.4%) and others (5.1%). The respective composition of solvent firms is manufacturing (27.4%), wholesale and retail trade(24.8%), real estate (16.9%), construction (13.9%) and others (17.1%), including publishing, administration and defence, education and health (Chen et al. 2011). In our project we focus on the four largest industry sectors.

Table 1: Variables of the Creditreform database

Column	Value
ID	ID of each company
T2	Solvency status (solvent:0, insolvent:1)
Jahr	Year
VAR1	Cash and cash equivalents
VAR2	Inventories
VAR3	Current assets
VAR4	Tangible assets
VAR5	Intangible assets
VAR6	Total assets
VAR7	Accounts receivable
VAR29	Accounts receivable vs. affiliated companies
VAR8	Lands and buildings
VAR9	Equity (own funds)
VAR10	Shareholder loan
VAR11	Accrual for pension liabilities
VAR12	Total current liabilities
VAR13	Total longterm liabilities
VAR14	Bank debt
VAR15	Accounts payable
VAR30	Accounts payable vs. affiliated companies
VAR16	Sales
VAR17	Administrative expenses
VAR18	Amortization and depreciation
VAR19	Interest expenses
VAR20	Earnings before interest and taxes (EBIT)
VAR21	Operating income
VAR22	Net income
VAR23	Increase (decrease) in inventories
VAR24	Increase (decrease) in liabilities
VAR25	Increase (decrease) in cash
VAR26	Industry classification code
VAR27	Legal form
VAR28	Number of employees
Rechtskreis	Accounting principle
Abschlussart	Type of account

3 Data preparation

For cleaning the data we use primarily the `dplyr` package, which allows us to manipulate data easier than with base `R` allowing for less verbose, i.e. more easily readable code. In order to clean the data we load the Creditreform dataset into `R` with the `read.csv()` command and store it as `data`. Using `dplyrs`'s `filter()` command we choose only those observations from the years between 1997-2002 and store them as `data1`.

Listing 1: |**data.preparation.R**|

```r
data= read.csv("Data/SPL_data.csv", sep = ";", dec = '.', header = TRUE,
               stringsAsFactors = TRUE)

# Due to missing insolvencies in 1996 and missing data from 2003 onwards,
# we choose only the data of the period 1997-2002
#data1 = data 1997-2002
data1 = filter(data, JAHR>= 1997 & JAHR<=2002)
```

We then want to choose only those observations belonging to the four most prevalent industry sectors in both solvent and insolvent firms, i.e manufacturing, wholesale and retail trade, real estate and construction. We extract the industry class of firms by using the `substring()` command and save it in a new column of `data1`.

Listing 2: |**data.preparation.R**|

```r
data1$Ind.Klasse = substring(data1$VAR26, 1, 2)
```

These two digits contained in the variable `Ind.Klasse` are then used to identify in which of the 17 broad industry sectors, defined in the German Classification of Economic Activities Edition 1993 (WZ93), issued by the German Federal Statistical Office (destatis.de), each firm belongs. If the value of the variable is in the range 15-37 then the firm belongs to the manufacturing sector. Accordingly the ranges 50-52 and 70-74 correspond to "Wholesale and Retail Trade" and "Real Estate" respectively, while the value 45 corresponds to "Construction". We create four subsets of data1 for each of the above mentioned sectors by using the `filter()` command again and remove `data` and `data1` with the `rm()` command as we do not need them anymore. The 4 subsets are then bound with `rbind()` into a new dataset with the name `data`.

Listing 3: |**data.preparation.R**|

```r
Man = filter(data1, Ind.Klasse %in% as.character(15:37))
Man$Ind.Klasse = "Man"

WaR = filter(data1, Ind.Klasse %in% as.character (50:52))
WaR$Ind.Klasse = "WaR"

Con = filter(data1, Ind.Klasse == "45")
Con$Ind.Klasse = "Con"

RE = filter(data1, Ind.Klasse %in% as.character(70:74))
RE$Ind.Klasse = "RE"

# Remove data and data1 & bind the above subsets to get one dataset containing
# only companies of interest

rm(data, data1)

data = rbind(Man, WaR, Con, RE)
```

Table 2: Number of solvent and insolvent companies per year in the dataset

Year	Solvent	Insolvent
1997	1084	126
1998	1175	114
1999	1277	147
2000	1592	135
2001	1920	132
2002	2543	129

We then turn our interest on the size of the companies. Specifically the distribution of total assets which can be considered to be representative of the distribution of the companies' size (Chen et al., 2011). Following the methodology of Zhang & Härdle (2010), we keep only firms with total assets (`VAR6` in the dataset) in the range of $10^5 - 10^8$ Euros, since the credit quality of small firms often depends mostly on the finances of a key individual (e.g. the owner).

Finally, we eliminate observations with zero values in variables used as denominators in the calculation of the financial ratios that will be used for the classification of the companies (lines 60-67 in the code) and save the result as `data_clean`.

Listing 4: |**data.preparation.R**|

```
60 data_clean = data %>% filter(VAR6 != 0,
61                              VAR16 != 0,
62                              VAR1 != 0,
63                              VAR2 != 0,
64                              VAR12 != 0,
65                              VAR12 + VAR13 != 0,
66                             VAR6 - VAR5 - VAR1 - VAR8 != 0,
67                             VAR19 != 0)
```

We end up with 9591 solvent and 783 insolvent firms, which is a similar result as in Chen et al. (2010).

3.1 Financial Ratios

The Creditreform database contains many financial statement variables for each company. Such statements are often used by investors to evaluate firms in two basic ways. A firm is either compared with itself by analysing how it has changed over time, or the firm is compared to other similar firms by means of a common set of financial ratios (Berk & DeMarzo, 2016). We follow the methodology of Chen et al. (2011) and use 23 financial statement variables to create 28 financial ratios to be used in classification. The variables used in the creation of the financial ratios are summarized in table 2. These financial ratios can be divided into six main groups (risk factors): profitability, leverage, liquidity, activity, firm size and percentage change for some variables.

Table 2 presents all the financial ratios used in the current study, the formulas used for their calculation and their category. In our code, the calculation procedure of the financial ratios can be found in lines 92-121 where we add the ratios for each firm to the dataset by using the `dplyr`'s `mutate()`-command. Then we create a dataset `test_data_rel` where only relevant variables are kept, i.e. the ID of the firm, it's solvency status, the year as well as the 28 financial ratios.

Profitability ratios have appeared in many studies to be strong predictors for bankruptcy (Chen et al., 2011). They measure the ability of a firm to generate revenue relative to its costs over a specific

Table 3: Variables used for the calculation of financial ratios and their description

Variable	Description	Variable	Description
VAR1	Cash and cash equivalents	VAR14	Bank debt
VAR2	Inventories	VAR15	Accounts payable
VAR3	Current assets	VAR16	Sales
VAR5	Intangible assets	VAR18	Amortization and depreciation
VAR6	Total assets	VAR19	Interest expenses
VAR3 - VAR2	Quick assets	VAR20	EBIT
VAR7	Accounts receivable	VAR21	Operating income
VAR8	Lands and buildings	VAR22	Net income
VAR9	Equity (own funds)	VAR23	Increase (decrease) inventories
VAR12	Total current liabilities	VAR24	Increase (decrease) liabilities
VAR12 + VAR13	Total liabilities	VAR25	Increase (decrease) cash
VAR3 - VAR12	Working capital		

time period. We calculate 7 ratios belonging to this group (ratios x1-x7). The return on assets ratio (x1), for example, provides information on how effective a firm is in making use of its assets to create income. A higher ratio signals that a firm is able to earn more money on less investment (Chen et al., 2011).

Among the profitability ratios, the net profit margin ratio `x2` shows the percentage of sales which the firm keeps in earnings. A high ratio corresponds to a firm with more profitability and better control over its costs (Chen et al., 2011).

Listing 5: |**data.preparation.R**|

```
test_data = data_clean %>%
  mutate(x1 = VAR22/VAR6,
         x2 = VAR22/VAR16,
         x3 = VAR21/VAR6,
         x4 = VAR21/VAR16,
         x5 = VAR20/VAR6,
         x6 = (VAR20+VAR18)/VAR6,
         x7 = VAR20/VAR16,
         x8 = VAR9/VAR6,
         x9 = (VAR9-VAR5)/(VAR6-VAR5-VAR1-VAR8),
         x10 = VAR12/VAR6,
         x11 = (VAR12-VAR1)/VAR6,
         x12 = (VAR12+VAR13)/VAR6,
         x13 = VAR14/VAR6,
         x14 = VAR20/VAR19,
         x15 = VAR1/VAR6,
         x16 = VAR1/VAR12,
         x17 = (VAR3-VAR2)/VAR12,
         x18 = VAR3/VAR12,
         x19 = (VAR3-VAR12)/VAR6,
         x20 = VAR12/(VAR12+VAR13),
         x21 = VAR6/VAR16,
         x22 = VAR2/VAR16,
         x23 = VAR7/VAR16,
         x24 = VAR15/VAR16,
         x25 = log(VAR6),
         x26 = VAR23/VAR2,
         x27 = VAR24/(VAR12+VAR13),
         x28 = VAR25/VAR1)
```

Another important factor of risk measurement is leverage. It refers to the extent that a firm relies on debt as a source of financing (Berk & DeMarzo, 2016). As firms combine debt and equity to

Table 4: Definitions of financial ratios

Ratio No.	Formula	Ratio	Category
x1	VAR22/VAR6	Return on assets (ROA)	Profitability
x2	VAR22/VAR16	Net profit margin	Profitability
x3	VAR21/VAR6		Profitability
x4	VAR21/VAR16	Operating profit margin	Profitability
x5	VAR20/VAR6		Profitability
x6	(VAR20+VAR18)/VAR6	EBITDA	Profitability
x7	VAR20/VAR16		Profitability
x8	VAR9/VAR6	Own funds ratio (simple)	Leverage
x9	(VAR9-VAR5)/(VAR6-VAR5-VAR1-VAR8)	Own funds ratio (adjusted)	Leverage
x10	VAR12/VAR6		Leverage
x11	(VAR12-VAR1)/VAR6	Net indebtedness	Leverage
x12	(VAR12+VAR13)/VAR6		Leverage
x13	VAR14/VAR6	Debt ratio	Leverage
x14	VAR20/VAR19	Interest coverage ratio	Leverage
x15	VAR1/VAR6		Liquidity
x16	VAR1/VAR12	Cash ratio	Liquidity
x17	(VAR3-VAR2)/VAR12	Quick ratio	Liquidity
x18	VAR3/VAR12	Current ratio	Liquidity
x19	(VAR3-VAR12)/VAR6		Liquidity
x20	VAR12/(VAR12+VAR13)		Liquidity
x21	VAR6/VAR16	Asset turnover	Activity
x22	VAR2/VAR16	Inventory turnover	Activity
x23	VAR7/VAR16	Accounts receivable turnover	Activity
x24	VAR15/VAR16	Accounts payable turnover	Activity
x25	log(VAR6)		Size
x26	VAR23/VAR2	Percentage of incremental inventories	Percentage
x27	VAR24/(VAR12+VAR13)	Percentage of incremental liabilities	Percentage
x28	VAR25/VAR1	Percentage of incremental cash flow	Percentage

finance their operations, leverage ratios are useful in evaluating a firm's ability to meet its financial obligations. We calculate 7 ratios belonging to this group (x8-x14). An example of a leverage ratio is the net indebtedness x11 which measures the level of short term liabilities not covered by the firm's most liquid assets as a proportion to the firm's total assets. Except from measuring the short term leverage of a firm, this ratio provides a measure of liquidity as well. Another popular leverage ratio is the debt ratio x13, which is defined as the debt of a company divided by its total assets. While this ratio performs well for public firms, it performs considerably worse for private firms compared to the total liabilities to total assets ratio x12. The reason for that is that liabilities is a more inclusive term which includes debt, deferred taxes, minority interest, accounts payable and other liabilities (Chen et al.,2011).

The next six financial ratios we calculate belong to the family of liquidity ratios (x15-x20). Liquidity is a common variable in many credit decisions and represents a firm's ability to convert an asset into cash quickly (Chen et al.,2011). Liquidity ratios are important indicators of a firm's health as they assess its ability to meet its debt obligations. Chen et al. (2011) note that the cash to total assets ratio x15 is the most important single variable relative to default in the private dataset. The quick ratio x17 is an indicator used to assess if a firm has adequate liquidity to meet short term needs. A higher quick ratio indicates that a cash shortfall of the firm is less likely to occur in the near future (Berk & DeMarzo, 2016).

Another type of ratios which deliver important information on insolvency are the activity ratios x21-x24. They measure the efficiency of a firm in using its own resources to generate cash and revenue. The asset turnover ratio x21 measures the ability of a company to produce sales from its assets by comparing sales to its asset base. Accounts receivable x23 and accounts payable x24 turnover ratios are powerful predictors (Chen et al., 2011).

Additionally, we compute a risk indicator accorting to the size of each firm, which is defined as the logarithm of total assets x25 in order to study the insolvency risk of small, medium and large

Table 5: Three number summary of the financial ratios for solvent and insolvent firms.

	Insolvent			Solvent		
Ratio	$q_{0.05}$	Median	$q_{0.95}$	$q_{0.05}$	Median	$q_{0.95}$
x1	-0.19	0.00	0.09	-0.09	0.02	0.19
x2	-0.15	0.00	0.06	-0.07	0.01	0.09
x3	-0.22	0.00	0.10	-0.11	0.03	0.27
x4	-0.16	0.00	0.06	-0.08	0.02	0.13
x5	-0.09	0.02	0.13	-0.09	0.05	0.27
x6	-0.13	0.07	0.21	-0.04	0.11	0.35
x7	-0.14	0.01	0.10	-0.07	0.02	0.14
x8	0.00	0.05	0.40	0.00	0.14	0.60
x9	-0.01	0.05	0.56	0.00	0.16	0.96
x10	0.18	0.52	0.91	0.09	0.42	0.88
x11	0.12	0.49	0.89	-0.05	0.36	0.83
x12	0.29	0.76	0.98	0.16	0.65	0.96
x13	0.00	0.21	0.61	0.00	0.15	0.59
x14	-7.75	1.05	7.19	-6.76	2.16	74.37
x15	0.00	0.02	0.16	0.00	0.03	0.32
x16	0.00	0.03	0.43	0.00	0.08	1.41
x17	0.18	0.68	1.88	0.24	0.94	4.55
x18	0.57	1.26	3.72	0.64	1.58	7.15
x19	-0.32	0.15	0.63	-0.22	0.25	0.73
x20	0.34	0.84	1.00	0.22	0.86	1.00
x21	0.24	0.61	2.31	0.16	0.48	2.01
x22	0.02	0.16	0.88	0.01	0.11	0.56
x23	0.02	0.12	0.33	0.00	0.09	0.25
x24	0.03	0.14	0.36	0.01	0.07	0.23
x25	13.01	14.87	17.16	12.82	15.41	17.95
x26	-1.20	0.00	0.74	-0.81	0.00	0.57
x27	-0.44	0.00	0.47	-0.53	0.00	0.94
x28	-12.17	0.00	0.94	-7.03	0.00	0.91

firms (Chen et al., 2011). Finally, we calculate the ratios of the percentage change of incremental inventories, liabilities and cash flow (`x26-x28`). As the increased cash flow is the additional operating cash flow that an entity receives from taking on a new project, a positive incremental cash flow means that the firm's cash flow will increase with the acceptance of a project, the ratio of which indicates that the firm should invest time and money in the project (Chen et al., 2011).

Listing 6: |**data.preparation.R**|

```r
80 replace_extreme_values = function(x, ...){
81   # The point of this is to allow more flexibility for the user.
82   quantiles = quantile(x, ...)
83 
84   x = ifelse(x <= quantiles[1], quantiles[1], x)
85   x = ifelse(x >= quantiles[2], quantiles[2], x)
86 
87   return(x)
88 }
```

In order to avoid sensitivity to outliers in applying the Random Forest, CART and the logit model, we follow the methodogy of Chen et al. (2011) and we replace extreme ratio values according to the following rule: For i = 1, ..., 28, if $x_i < q_{0.05}(x_i)$, then $x_i = q_{0.05}(x_i)$, and if $x_i > q_{0.95}(x_i)$, then $x_i = q_{0.95}(x_i)$, where $q_{0.05}(x_i)$ and $q_{0.95}(x_i)$ refer to the 0.05 and 0.95 quantiles of the ratio x_i respectively. This will make our results robust and insensitive to outliers. For that purpose we create the function `replace_extreme_values()` (lines 80-88), which is then separately applied to the subsets of solvent and insolvent companies. The lower and upper quantile as well as the median of the financial ratios for solvent and insolvent firms are presented in table 3.1. The final clean dataset to be used in further analysis is then created by binding the two subsets of solvent and insolvent firms and is saved as `data_clean`. The resulting dataframe coincides almost entirely with the results reported by Chen et al. (2011).

4 Linear Discriminant Analysis

4.1 Theory

Since we are mostly interested in classifying if a company is going to default or not, we opted for a method that is especially suited to separate labelled data. Linear Discriminant Analysis (LDA) is a technique for dimensionality reduction that encorporates information on class-labels of the different observations. In contrast to Principal Component Analysis, which is a unsupervised dimensionality reduction technique, it finds the rotation that ensures the highest separability between classes. It accomplishes this goal by trying to maximize between class variance while simultaneously minimizing the within class variance (Härdle, 2003).

$$max\ J_b(w) = w\prime S_b w, \qquad w \in \mathbb{R}^d, d \in \mathbb{N} \tag{1}$$

$$min\ J_w(w) = w\prime S_w w \tag{2}$$

This is done by maximizing the so called Rayleigh coefficient

$$max J = \frac{J_b(w)}{J_w(w)} = \frac{w\prime S_b w}{w\prime S_w w}. \tag{3}$$

The matrices for between and within class variance are defined as

$$S_b = \sum_{c=1}^{C} (\mu_c - \mu)(\mu_c - \mu)\prime \tag{4}$$

$$S_w = \sum_{c=1}^{C} \sum_{i \in c} (x_i - \mu_c)(x_i - \mu_c)\prime \tag{5}$$

where C is the number of classes, μ_c is the vector of sample means for each class respectively and μ is the vector of sample means for the full dataset. For identification purposes we can always chose weights w such that $w\prime S_w w = 1$, since J is constant with regards to rescalings. We can therefore replace w by αw which will result in the constant α canceling out. This way the initial optimization problem can be formulated as

$$\arg\min_w -\frac{1}{2} w\prime S_b w \quad s.t. \quad w\prime S_w w = 1 \tag{6}$$

with the lagrangian being

$$\mathcal{L} = -\frac{1}{2} w\prime S_b w + \frac{1}{2}\lambda \left(w\prime S_w w - 1 \right). \tag{7}$$

The halves are added for more convenient matrix derivatives. The Karush-Kuhn-Tucker conditions imply that the solution to this maximization problem and subsequently the vector of weights we want to find needs to fulfill

$$S_b w = \lambda S_w w. \tag{8}$$

Which results in a generalized eigenvalue problem (Duda, 1973). For solving these, there exists a convenient `R`-solution in the form of the `geigen`-package.

4.2 Implementation

The result from 8, which is essentially the rotation of the underlying data's column space that ensures the highest separability, is implemented in the `lda`-class for the two-class case. The `lda()` function accepts two arguments:

1. `data`: a `data.frame` containing at least one column of factors indicating the class label.
2. `by`: a `character`-string equal to the column's name containing the class labels. Note that all other non-numerical columns will be ignored by the function, since LDA is only meaningful for continuous variables.

The function in a first step extracts the useable columns and the number of classes provided in `data`. It then performs a quick check if the prerequisites are met and then continues with the calculation of the class-means μ_1 and mu_2, the overall mean μ, as well as the scatter-matrices S_b and S_w. With these we can solve the generalized eigenvalue problem from 8.

Algorithm 1 `lda()`

```
 1: procedure SET UP AUXILIARY VARIABLES
 2:     $num$ ← extract numeric columns
 3:     $classes$ ← extract class labels
 4:     $mu$ ← calculate class means
 5:     $x\_bar$ ← calculate overall means
 6:     $S$ ← calculate variance per group
 7:     $S\_b1$ & $S\_b2$ ← get summands for between class scatter matrix
 8: procedure CALCULATE SCATTER MATRICES
 9:     $S\_b$ ← S_b1 + S_b2
10:     $S\_w$ ← S[[1]] + S[[2]]
11: procedure SOLVE GENERALIZED EIGENVALUE PROBLEM
12:     $V$ ← calculate Eigenvectors and Eigenvalues
13:     $v$ ← extract 2 Eigenvectors associated to largest Eigenvalues
14:     $lda1$ & $lda2$ ← calculate first two LDA-Components
15:     $inertia$ ← calculate percentage of explained variance
16:     return: lda1, lda2, classes, mu, v, inertia
```

The first task of the `lda()`-function is to determine, if the prerequisites for further computation are met. To this avail it first asserts that the user provided a dataframe that contains at least one numeric column and a column that contains exactly two distinct class labels (lines 11-17). In a second step it calculates all required variables such as the class means μ_1 and μ_2, the overall mean μ and the covariance matrices for each group (lines 18-19). The class specific metrics are computed by calls to custom made functions defined in the `utils.R`-file of the `LDA`-directory. Essentially these functions are wrappers for subsetting a provided dataframe by a provided key, here this is the class-label, and and an `apply` call looping over the respective subset's columns. The `get_class_means()`

calculates the mean inside the `apply`-function whereas `get_class_cov()` uses a call to the built-in `cov`-function. However note that `cov` calculates

$$\hat{cov}(X) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})(x_i - \bar{x})\prime.$$

Unfortunately this is not exactly what we want for the scatter matrix defined in 5. We need to multiply the resulting list of covariance matrices elementwise by the number of observations in each class minus one. We correct this before calculating the within class scatter matrix on line 31.

Listing 7: |**lda.R**|

```
6  lda = function(data, by){
7    # Closed form solution of LDA:
8    # w = S_b^-0.5 * largest_eigenvector(S_b^0.5 * S_w^-1 * S_b^0.5)
9  
10   # Check prerequisites
11   num = get_numeric_cols(data = data)
12   classes = unique(data[, by])
13 
14   stopifnot(
15     length(num) > 0,
16     length(classes) == 2)
17 
18   mu = get_class_means(Data = data, By = by, na.rm = TRUE)
19   S = get_class_cov(Data = data, By = by, use = "complete.obs")
20   n = sapply(classes, function(x){sum(data[, by] == x)})
21   print(n)
22   # Compute overall mean:
23   x_bar = colMeans(data[, num], na.rm = TRUE)
24 
25   # Compute between class scatter matrix:
26   Sb1 = (mu[, 1] - x_bar) %*% t(mu[, 1] - x_bar)
27   Sb2 = (mu[, 2] - x_bar) %*% t(mu[, 2] - x_bar)
28   S_b = Sb1 + Sb2
29 
30   # Compute within class scatter matrix:
31   S_w = (n[1]-1)*S[[1]] + (n[2]-1)*S[[2]]
```

With the `matrix`-object `S_w` storing the within-scatter-matrix S_w and `S_b` storing S_b we can continue by solving the generalized Eigenvalue problem posed in 8. This is done by the `geigen()` function from the `geigen` package.

Listing 8: |**lda.R**|

```
36   V = geigen(S_b, S_w, symmetric = TRUE)
37 
38   # Extract (absolutely) largest eigenvalue
39   ev_order = order(abs(V[["values"]]), decreasing = TRUE)
40   v = V[["vectors"]][ev_order[1:2], ]
41 
42   # Percent of variance explained:
43   inertia = (V[["values"]][ev_order[1:2]])^2 / sum(V[["values"]]^2)
```

We extract the eigenvectors and eigenvalues from `V` (line 36) in form of a list. However, caution is required because the eigenvectors are not ordered, which is different to the implementation of eigenvalues in the `base`-function `eigen()`. Therefore we rearrange the eigenvectors according to their absolute eigenvalues (line 39). We extract only the first two eigenvectors, because we require the most informative rotations in order to produce two-dimensional plots. Additionally we compute the percentage of explained variance as

$$intertia_i = \frac{\lambda_i^2}{\sum_{j=1}^{d} \lambda_j^2}. \tag{9}$$

The rotations `lda1` and `lda2` are computed by an `apply`-call (lines 45-46) to an anonymous function which emulates a scalar product by multiplying the two vectors `v[, i]`, which is the i-th eigenvector $i \in \{1, 2\}$, with each observation in the provided dataset. Computationally this is equivalent to the matrix multiplication Xv_i, where X is the data matrix and v_i is the eigenvector.

Finally we gather the results into a `list` and sets it's class to `flda`. This allows us to augment pre-existing functions with a custom method for predicting and plotting for future objects of the`flda`-class in an object-oriented fashion.

4.2.1 Predictions

To make predicting with the `flda`-class as easy as plotting we augment the standard `predict` function in a similar fashion. We expect the user to have already trained an `flda`-model to which she or he wishes to apply new data and obtain predictions on the class labels.

Listing 9: |**lda.R**|

```r
71 predict.flda = function(model, data){
72   v = model$scalings[1, ]
73   num = get_numeric_cols(data)
74 
75   Data = data[, num]
76   dims = dim(Data)
77   stopifnot(dims[2] == length(v))
78 
79   # Compute discriminant as the dot product of every observation
80   # with the scaling vector v:
81   discr = apply(X = Data, MARGIN = 1, FUN = function(x){sum(x * v)})
82 
83   # Compute cutoff threshold:
84   c = 0.5 * t(v) %*% (rowSums(model$class_means))
85   predictions = ifelse(discr <= as.numeric(c), model$classes[1], model$classes[2])
86 
87   return(predictions)
88 }
```

The provided new data should have a similar structure to the training data. This means that especially the order of the columns should be the same. We check for obviously non-conforming data in line 77 but we do not perform explicit tests on the order of columns. If the data has the required shape we calculate the discriminant function for each observation, which is the scalar product of the observation's data vector and the first eigenvector from the trained model. We then proceed calculating a threshold c for being able to discriminate between the two groups by following the rule

$$c = \frac{1}{2} v\prime (\mu_1 + \mu_2) \tag{10}$$

predictions are made according to the threshold, if the value of the discriminant function is smaller than the threshold we assign the label of the first class and the second class otherwise (Duda, 1973).

4.2.2 Plotting

With a trained model of the `flda`-class it may be of interest to plot the rotations in order to get a visualisable idea of the separability of the two classes. To make this as easy as possible for the end-user we implemented a `plot` method for each instance of a `flda`-class. This means that a user can simply store a call to the `lda`-function inside a variable, called `model` for example. To plot the rotations one can simply invoke `plot` on `model` as easy as with any other model class by typing `plot(model)`.

This is done by augmenting `R`'s `plot`-function using the `plot.__className__` syntax. Here we replace `__className__` by `flda`

Listing 10: |**lda.R**|

```r
plot.flda = function(model){
  p = ggplot2::ggplot(data = model$X, ggplot2::aes_string(x = "lda1", y = "lda2",
    color = "labels")) +
    ggplot2::geom_point(alpha = 0.5, shape=21) +
    ggplot2::ggtitle("Projection on the first 2 linear discriminants") +
    ggplot2::xlab("First linear discriminant") +
    ggplot2::ylab("Second linear discriminant") +
    ggplot2::theme_bw()
  return(p)
}
```

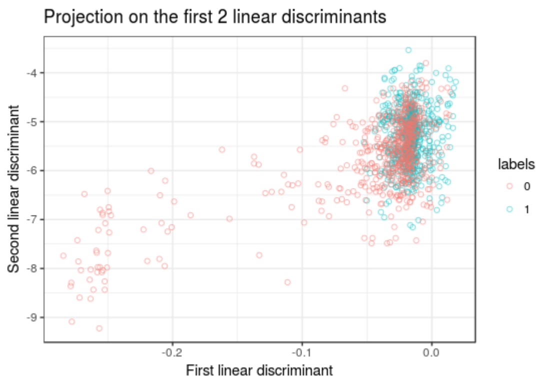
We overwrite the behavior of `plot` in order to accomodate `flda`-objects. Here we pass the model's precomputed rotations to the `ggplot2` function. The `ggplot2::__function__` makes it explicit that we want the `__function__` from the `ggplot2` package. Another benefit of this syntax is that this way we do not implicitly change a users namespace, which may potentially hide some functions by overwriting them with `ggplot2` routines and therefore may result in unexpected behavior.

Looking at the creditreform dataset we can apply our custom routines by creating a train-set first and then visualizing the results. This reveals already why LDA's predictive results are of poor quality. The feature space spanned by the financial ratios is simply not linearly separable.

Listing 11: |**creditLDA.R**|

```r
creditLDA = lda(data = train,
                by = "T2")
plot(creditLDA)
```

Figure 1: Using Linear Discriminant Analysis on the Creditreform Database



However, we can see here, that it is not possible to separate the two classes, especially by linear techniques. The plot visualises the LDA's inability to provide satisfyingly accurate predictions (Härdle et al., 2012).

5 Binary Response Models

5.1 Theory

Logit and Probit models are the most important and probably most used member of the class of models called Binary Response or Generalized Linear Models. These parametric models are applied to estimate the probability of an observation belonging to a particular class while using observed data of known outcomes to infer models which predict the outcome of future observations.

The outcome of interest in the Creditreform-dataset is of binary nature. Firms can either go bankrupt or not. This behavior is commonly modeled by Binary Response Models like the probit or the logit model. Binary response variables follow a Bernoulli probability function

$$f(y|x) = P(y=1|x)^y \left(1 - P(y=1|x)\right)^{1-y}, \qquad y \in \{0,1\},\ x \in \mathbb{R}^d, d \in \mathbb{N}, \tag{11}$$

where $P(y=1|x)$ stands for the conditional probability of observing $y=1$ given x. Both probit and logit models have in common that $P(y=1|x)$ is modeled by a monotonic transformation of a linear function

$$P(y=1|x) = G(x\prime\beta), \qquad \beta \in \mathbb{R}^d, \tag{12}$$

where $x\prime\beta$ is the scalar product of x and β. Additionally we require that $0 \leq G(x\prime\beta) \leq 1$, since it denotes a probability.

For the probit model G will be the cumulative density function of the normal distribution

$$P(y=1|x) = G(x\prime\beta) = \Phi(x\prime\beta) = \int_{-\inf}^{x\prime\beta} \frac{1}{\sqrt{2\pi}} exp\left[-(\frac{t^2}{2})\right] dt. \tag{13}$$

Here $\Phi(x\prime\beta)$ stands for the cumulative density function of the normal distribution.

For the logit model G will be replaced by the cumulative density function of the logistic distribution $\Lambda(x\prime\beta)$:

$$P(y=1|x) = G(x\prime\beta) = \Lambda(x\prime\beta) = \frac{exp(x\prime\beta)}{1 + exp(x\prime\beta)} \tag{14}$$

The parameter vector β is obtained by the Maximum-Likelihood method. Given independent and identically distributed samples, the Likelihood function can be written as

$$\begin{aligned} L(\beta; y, x) &= \prod_{i=1}^{n} f(y_i|x_i) = \prod_{i=1}^{n} P(y_i = 1|x_i)_i^y \left(1 - P(y_i=1|x_i)\right)^{1-y_i} \\ &= \prod_{i=1}^{n} G(x_i\prime\beta)_i^y \left(1 - G(x_i\prime\beta)\right)^{1-y_i} \end{aligned} \tag{15}$$

where we just take the product over all individual Bernoulli-functions.

The log-Likelihood can thus be written as

$$l = \log L(\beta; yx) = \sum_{i=1}^{n} y_i \log G(x_i\prime\beta) + (1-y_i)\log(1 - G(x_i\prime\beta)) \tag{16}$$

The Maximum-Likelihood estimators $\beta_M L$ are calculated as

$$\beta_M L = argmax(l) \tag{17}$$

and solve the first order conditions for a maximum.

$$\frac{\partial l}{\partial \beta} \overset{!}{=} 0 \tag{18}$$

In general, the resulting system of equations has no closed-form solution for $\beta_M L$ and numerical solutions are needed which can be obtained by iterative optimization techniques (Winkelmann 2009). We will implement one of these methods in the next section.

5.2 Implementation

Our aim is to provide a function class which allows the end user to easily train a Binary Response Model of the *logit* or *probit* variety providing a dataset with an observed outcome variable. After training the user should also be in a position, where it is equally easy to obtain predictions for new data. With this goal in mind we created the `brm`-class. The architecture of the `brm`-class follows the general structure outlined in the chapter before. First, it generates a log-Likelihood-function, which it then optimizes using a Gradient-Descent-Algorithm. After training the model it is possible for the user to obtain predictions by invoking the `predict()`-function, which has been augmented with a method for the `brm`-class.

5.2.1 Obtaining the Likelihood

The first task is to define a function that accepts a distribution and yet undefined data as it's input and first extracts all suitable variables, then expresses the Likelihood-function from 16 and finally returns another function which depends only on the weights β. This task is performed by the `get_likehood()`-function. We outline the function pass of `get_likehood()` first in pseudo-code followed by a look on the implementation in the `R` language.

Algorithm 2 get_likehood()

```
1: procedure SET UP AUXILIARY VARIABLES
2:     $grp \leftarrow$ unique labels
3:     $nums \leftarrow$ extract numeric columns
4:     $Xy\_mat \leftarrow$ bind numeric variables as a matrix
5:     $y\_pos \leftarrow$ cache position of the outcome variable
6: procedure SET UP LOG-LIKELIHOOD
7:     $l \leftarrow 0$
8:     for: $x_i, y_i$ in data :
9:         calculate : $j = y_i \log G(x_i\prime\beta) + (1 - y_i) \log(1 - G(x_i\prime\beta)$
10:        update : $l \leftarrow l + j$
11:    return : $l(\beta)$
```

The heavy lifting in this function is done by this R-snippet:

```r
l = function(x){
  - sum(apply(Xy_mat, 1,
          function(X){
            X[y_pos] * distr(t(X[-y_pos]) %*% x,
                             lower.tail = TRUE,
                             log.p = TRUE) +
              (1-X[y_pos]) * distr(t(X[-y_pos]) %*% x,
                                   lower.tail = FALSE,
                                   log.p = TRUE)})))}
# Return loglikelihood as a function of x (here 'x' stands for the weights)
return(l)
```

The for-loop from the pseudo code is implemented as an apply-call to Xy_mat, which in turn is a matrix of numeric columns. The apply-function initially selects each row in Xy_mat and extracts the outcome-variable Xy_mat[y_pos] which corresponds to y_i from 16. It then computes the scalar product between the regressors of Xy_mat's row, which plays the role of $x_i\prime\beta$ in 16. The scalar product is wrapped in dist which represents the cumulative distribution function $G(x\prime\beta)$ and is one of the arguments to get_loglikelihood().

Listing 12: |**logit.R**|

```r
distr = switch (mode,
                "logit" = plogis,
                "probit" = pnorm
)
```

This way dist will point to the built-in functions for computing probabilities, depending if the user wishes to train a logit or a probit model. The arguments lower.tail=TRUE and lower.tail=FALSE stand for $G(x\prime\beta)$ and $G(x\prime\beta) = 1 - G(x\prime\beta))$ respectively. Finally the resulting vector is summed up and multiplied by -1. This is done because of the way we implemented the Gradient Descent algorithm. Currently gradientDescentMinimizer() can only find minima. However, Maximum-Likelihood estimation poses a maximization problem. Luckily, we can transform any maximization problem into a minimization problem by multiplying with minus one.

5.2.2 Gradient Descent

Since we now have a log-Likelihood function, the next step is to optimize it. To this effect we deploy a Gradient Descent algorithm. Theory tells us that in order to reach the minimum of a function $f(x)$ starting at a particular $x \in \mathbb{R}^d, d \in \mathbb{N}$ one needs to follow the negative gradient $\nabla f(x)$ of f evaluated at x. This leads to the iterative rule we can exploit

$$x_{t+1} = x_t - \eta \cdot \nabla f(x_t), \qquad t \in \mathbb{N}, \eta \in \mathbb{R}^+ \tag{19}$$

where η is the learning rate (Duda, 1973). To this standard method of performing a Gradient Descent routine we will also make some minor modifications. First of all, we will approximate the gradients by taking finite differences, which is easier to implement albeit computationally inefficient. Finite differences are computed by

$$\nabla f(x_t) \approx \frac{f(x_{t+1}) - f(x_t)}{\epsilon}, \qquad \epsilon > 0 \tag{20}$$

Secondly, we will before we initialize the algorithm, try a set of random points and chose the one that provides the lowest value of the objective function as a starting point for the Gradient Descent Routine. This also ensures to an extend that the algorithm, if it reaches convergence, finds the global minimum. The final modification will be to prune the gradients. When computing gradients using finite differences, it may happen that the gradient's values can become extremely high for large denominators and very small ϵ. In fact, they can become high enough for `R` to treat them as `Inf` which results in the gradients being treated as `NaN` (not a number). To counteract that, we will limit the gradients to the interval $[-100, 100]$.

Algorithm 3 gradientDescentMinimizer()

```
1:  procedure SET UP AUXILIARY VARIABLES
2:      $learn\_rates \leftarrow$ descending sequence from learn to 0
3:      $a \leftarrow$ matrix of 1000 randomly initialized points
4:      $f\_a \leftarrow$ vector of function values for each element of a
5:      update: $a \leftarrow$ argmin(f_a)
6:      $gradient \leftarrow$ compute gradient evaluated at a
7:      $i \leftarrow 0$
8:  procedure PERFORM GRADIENT DESCENT
9:      $l \leftarrow 0$
10:     while: $i \leq max\_iter$ and any element of gradient $> 0$ :
11:         update : $a = a -$ learn_rates[i] $\cdot$ $gradient$
12:         calculate : $gradient =$ calculate gradient
13:     if i = max_iter then raise warning
14:     return : $a$
```

The `gradientDescentMinimizer()`-function accepts following arguments:

1. `obj`: an objective function, that accepts exactly one argument called 'x'.
2. `n_pars`: an integer specifying the dimensions of the objective.
3. `epsilon_step`: a float defining the stepwidth used for computing the finite differences.
4. `max_iter`: an integer for the maximum number of iteration before the algorithm aborts.
5. `precision`: a float defining the precision of the solution. All elements of the gradient have to be absolutely lower than `precision` for the algorithm to converge.
6. `learn`: a positive float representing the learning rate.
7. `verbose`: a boolean indicating if additional information during training is desired. The default is `FALSE`
8. `report_freq`: If `verbose` is `TRUE`, define how often to print the logstring. The default is 10 which corresponds to a console output being printed every 10 steps.

Listing 13: |**utils.R**|

```r
  a = matrix(data = runif(1000 * n_pars,
                          min = -10,
                          max = 10),
             ncol = n_pars)
  f_a = apply(a, 1, obj)
  a = a[which.min(f_a), ]
```

We begin by filling the matrix `a` with 1000 `n_pars`-dimensional points which we draw from the uniform distribution, making use of `R`'s built-in `runif`-function. We draw random numbers within the range of $[-100, 100]$ to cover a wide part of the objective function's domain.

Listing 14: |**utils.R**|

```r
130  get_gradient = function(x, d = n_pars,
131                          objective = obj,
132                          epsilon = epsilon_step){
133    init = matrix(data = x, nrow = d, ncol = d, byrow = TRUE)
134    steps = init + diag(x = epsilon, ncol = d, nrow = d)
135    f_steps = apply(steps, 1, objective)
136    f_comp =  apply(init, 1, objective)
137    D = (f_steps - f_comp) / epsilon
138    D_trimmed = ifelse(abs(D) <= 100, abs(D), 100) * sign(D)
139    return(D_trimmed)}
```

The workhorse in this routine is the `get_gradient()`-function, which computes the finite differences. First we need to compute the values of the objective function at the current and next step (lines 146 and 147). Then we can apply the current and next step as inputs to the objective function and compute the difference $f(x_{t+1}) - f(x_t)$. The current step is provided as the `x` argument to the function call. The next steps need to be inferred by the function. If $f(x_t)$ is multidimensional we need to perform an ϵ-step in each dimension of the vector, since we want to approximate the partial derivatives of $f(x)$ evaluated at x_t. I.e. first we want to increment just the first element of x and store the result, then just the second element, and repeat the process until we reach the last element. If we stack these vectors, we get a matrix of one-directional ϵ-steps that are essentially updates of the starting point x_t. The objective function is then evaluated at these updated points. The gradients are then computed as the element-wise difference of the updated point compared with the start, normalized by `epsilon_step`. The gradients are finally trimmed if necessary and returned.

Algorithm 4 get_gradient()

```
1: procedure SET UP AUXILIARY VARIABLES
2:     $$init \leftarrow \begin{bmatrix} x_1 & x_2 & \cdots \\ x_1 & x_2 & \cdots \\ x_1 & x_2 & \cdots \\ \vdots & \ddots & \cdots \end{bmatrix}$$
3:     $$steps \leftarrow \begin{bmatrix} x_1+\epsilon & x_2 & \cdots \\ x_1 & x_2+\epsilon & \cdots \\ x_1 & x_2 & \cdots \\ \vdots & \ddots & \cdots \end{bmatrix}$$
4:     $f\_comp \leftarrow$ apply row-wise: objective function to $init$
5:     $f\_steps \leftarrow$ apply row-wise: objective function to $steps$
6: procedure COMPUTE FINITE DIFFERENCES
7:     $$D \leftarrow \frac{f\_steps - f\_comp}{\epsilon}$$
8:     if $any d \in D \notin [-100; 100]$ then replace d by $100 \cdot \mathrm{sign(d)}$
```

In each iteration the `while`-loop ensures that convergence has not been reached. This is implemented by a call to `any` wrapped around a vector of logical expressions. If any element of the gradient is still greater than the specified precision, the call to `any` will evaluate to `TRUE`. The second breaking criterion is a safeguard for the loop not to run infinitely many times. If the current iteration is larger than `max_iter` the algorithm will break and the user will receive a warning (lines 175-176).

Listing 15: |**utils.R**|

```
155   while(any(abs(gradient) >= precision) & i <= max_iter){
156     if(i %% report_freq == 0 & verbose) {
157       cat("\nStep:\t\t", i,
158           "\nx:\t\t", a,
159           "\ngradient:\t", gradient,
160           "\nlearn:\t", learn_rates[i],
161           "\n-------------------------------------")}
162
163     i = i + 1
164     a = a - learn_rates[i] * gradient
165     gradient = get_gradient(a)
166   }
167
168   cat("\nResults\n",
169       "\nIteration:\t", i,
170       "\nx:\t\t", a,
171       "\nf(x):\t\t", obj(a),
172       "\ndf(x):\t\t", gradient,
173       "\n")
174
175   if(i >= max_iter){
176     warning("Maximum number of iterations reached.")}
177   return(a)
```

Should the user wish to receive information about the status of the algorithm during runtime, the optional argument `verbose` can be set to `TRUE` which will print a logstring to the console in regular intervals (lines 156-161). The results of the training are gathered together in the `out`-list. We store the calculated weights in `weights`, in `type` and `dist` we store wether a *logit* or a `probit`-model was used. `dist` contains a pointer to the `R`-cumulative density functions of the correct distribution. We also store the value of the log-Likelihood-function in `likelihood_val` for calculating test-statistics. Note however, that we did not implement this, since this paper focuses on obtaining predictions. Finally we assign the class and return `out`.

5.2.3 Predictions

In order to facilitate making predictions based on the `brm`-class we augmented the built-in function `predict()` with a method that works on our custom class in a predefined way.

Listing 16: |**logit.R**|

```
61 predict.brm = function(model, data){
62   Xb = as.matrix(cbind(constant = 1, data)) %*% model$weights
63   predictions = sapply(X = Xb,
64                        FUN = model$distribution,
65                        log.p = FALSE,
66                        lower.tail = TRUE)
67   return(predictions)
68 }
```

A call to `predict()` on a `brm`-model will add a column of ones to the provided `data`, to act as an intercept, and multiply the matrix with the weights calculated during training of the model (line

62). Finally these scores of the index-function $X\beta$ will be applied to the correct distribution. The distribution is stored inside `model$distribution` which points to `pnorm` in case of `brm`-model of `mode` "probit" and a pointer to `plogis` if the `mode` is equal to "logit".

5.3 Using the Logit-Model on the creditreform Dataset

We implemented a custom *logit*-model but since it is using only `R`-routines it can be slow on large datasets including a high number of variables. Instead, we use the `caret` package to build a logistic regression model. This package provides a `train()` method for fitting the data being able to chose among various algorithms. We chose the algorithm by specifying the `method` parameter in order to predict the target variable (Kuhn, et al., 2017).

Listing 17: |**logit_model_fin.R**|

```
glm_mod=train(as.factor(status)~.,data=training_complete, method="glm", family="
  binomial")
```

We apply the `glm` method. The function `glm()`, and the `predict` method for `glm` objects, reflect a classical approach to statistical inference. It only handles binary classification. Thus, we define our target variable `status` as categorical by using the `as.factor()` command (Stephens, 2016). All the other financial ratios are independent variables that influence the binary outcome of a target variable `status`. One important argument is `family` = `binomial`. The `family()` function specifies the assumed distribution of the dependent variable `status`. In our case, we model a target variable as a binomial distribution. The binomial family accepts the links such as "logit", "probit", "cauchit", "log" and "cloglog". The `link` function links the output to a linear model. It passes the dependent variable through the link function, and then models the resulting value as a linear function of the values of independent variables. Different combinations of family functions and link functions lead to different kinds of generalized linear models (for example, Poisson, or probit). Without an explicit `link` argument, the link function defaults to standard `logit` (Zumel & Mount, 2014). We store the resulting model in the object `glm_mod` and use `predict()` function to predict the insolvency of the German firms in the years 2000 until 2002. To this end, we pass the trained model `glm_mod` and the test set features without the actual labels to the `predict()` function. We specify the type of prediction as `type=prob` to receive the probability for a firm's insolvency. The predicted probabilities are passed into `evaluate_prediction` function to estimate the overall prediction power over models.

Listing 18: |**logit_model_fin.R**|

```
pred_logit=predict(glm_mod, newdata=validierung_logit,type="prob")
```

The practical advantage of logit model is that it provides insight into the impact of each predictor variable on the response variable. None of the nonparametric methods really provides this information. The coefficients of a logistic regression model can be thus treated as advice and may serve as orientation for decision making (Zumel & Mount, 2014). Yet, the calculation and interpretation of coefficients exceeds the scope of this project.

The logit model has some limitations. For example, if we include the wrong independent variables, the model will have little to no predictive value. It requires each data point to be independent of all other data points. If observations are related to one another, then the model will tend to overestimate the significance of those observations (Robinson, 2018). Further the method is searching for a single linear decision boundary that separates insolvent and solvent firms. This works well if the data are linearly separable. A linear separating boundary is, however, not suitable if there is doubt that the separation mechanism is of a nonlinear kind (Chen, et al., 2011). Therefore, it is always a good idea to try different models and choose the one which performs best on a test data set. Tree-based models for instance, divide the feature space into half-spaces using axis-aligned linear decision boundaries.

6 Classification and Regression Trees (CART)

Tree-based methods, or decision tree methods, are used for two broad types of problems - classification and regression. These methods are appropriate for extensive datasets. Its strength is that, in large data sets, it has the potential to reflect relatively complex forms of data structures, which may be hard to detect with conventional regression modelling. Additionally, the methodology is relatively easy to use and can be applied to a wide class of problems (Maindonald & Braun, 2010). Once again, we use the `caret` package to build a CART model. We pass the `rpart` method into the `train()` function to fit the data (Kuhn, et al., 2017). There is a package with same name `rpart`, which is specifically available for decision tree implementation. The `caret`-package links its train function with other packages simplifying the work process significantly.

Listing 19: |**cart_modell_fin.R**|

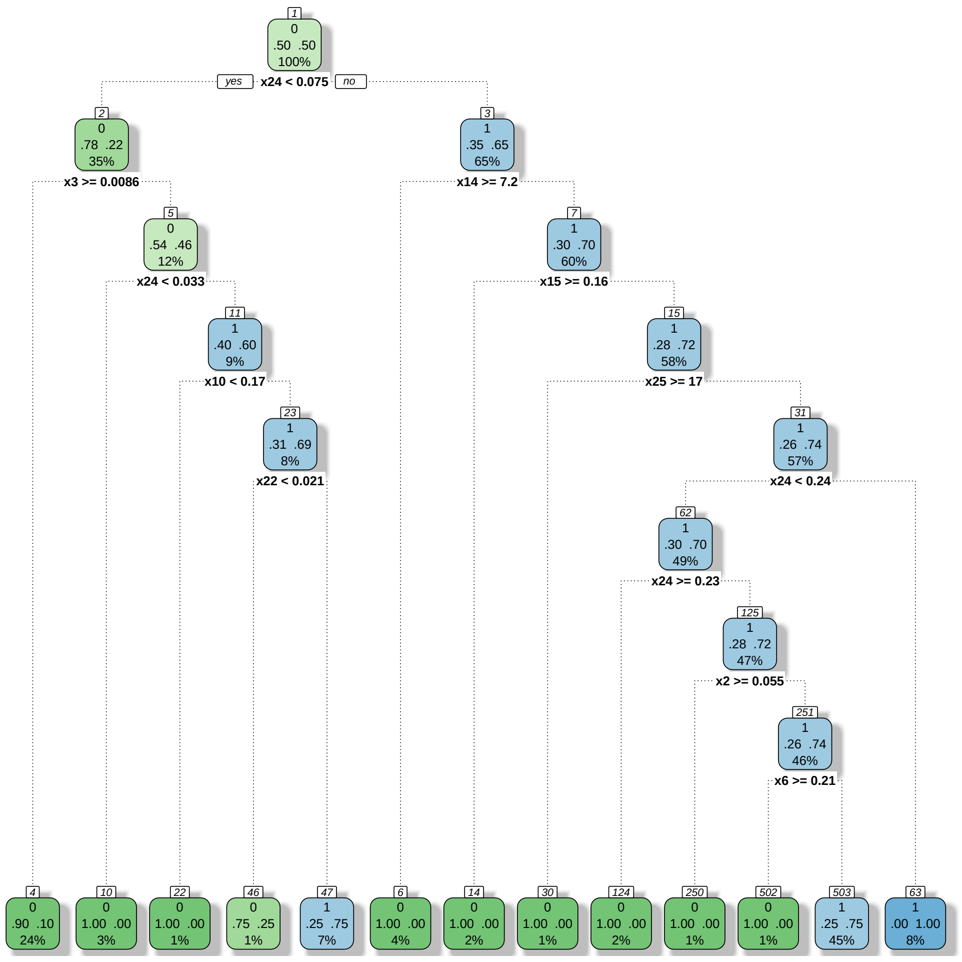
```
  modfit=train(status~.,method="rpart",data=training_complete)

  #calculate predictions and store the results in confusion matrix:
  pred.cart=predict(modfit,newdata=validierung_cart)
  conf_mat_cart=table(pred.cart,validierung_cart$status)
  TN=TN+conf_mat_cart[1,1]
  FP=FP+conf_mat_cart[1,2]
  TP=TP+conf_mat_cart[2,2]
  FN=FN+conf_mat_cart[2,1]
}
#calculate correlation matrix
correlations=round(cor(training_complete[, -1]),2)
pdf("CorrPlot.pdf")
corrplot(correlations, order = "hclust")
dev.off()
#perform variable importance
pdf("varImp.pdf")
plot(varImp(modfit), main="Variable Importance", top=10)
dev.off()

# use rpart for plotting a visually appealing decision tree and predictions
mod_fit=rpart(as.factor(status)~.,data=training_complete)

pdf("fancyRpartPlot.pdf")
fancyRpartPlot(mod_fit)
dev.off()

pred_cart=predict(mod_fit,newdata=validierung_cart,type="prob")
write.csv(cbind(label=as.numeric(as.character(validierung_cart$status)),pred_cart),
  file="CART/cart_pred.csv")
```

The basic way to visualize classification or regression tree built with R's `rpart()` function is to call `plot()`. The call typically produces a couple of black clouds of overlaid text. Therefore, the traditional representation of the CART model is not graphically appealing. There are better ways to plot `rpart()` trees for example by using the `prp()` function from `rpart.plot package` or the `fancyRpartPlot()` from `rattle` package (Milborrow, 2016). We have chosen the `fancyRpartPlot()` function to get a legible structure of our decision tree.

fancyRpartPlot() builds more elaborate and clean trees, which can be easily interpreted. Each node box displays the classification, the probability of each class at that specific node and the percentage of observations that arrived at each node (Rickert, 2013).

Once we have built the model, we are ready to validate it on a separate data set - the test set. We store the results in the object `mod_fit` and apply `predict()` function to calculate the probabilities for further analysis.

Figure 2: Decision Tree

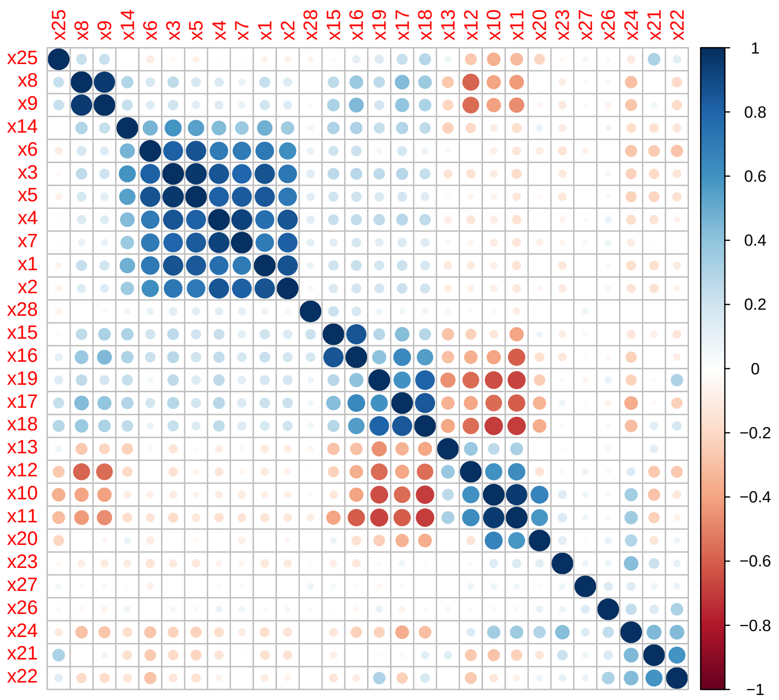


6.1 Variable Importance

It might be interesting to acquire information regarding which financial statement variables are most important in predicting the outcome. There are practical reasons for doing so. For example, we want to rank-order financial ratios in regard to their relative importance. This information can provide insights about the most important causes of insolvency, allow a deeper understanding of financial health of the companies and enable investors more effective decision making. If predictor variables were uncorrelated, this would be a simple task. We would rank the predictor variables by their correlation with the response variable. In our case though, the financial ratios variables are correlated (as shown in the chart below), and this complicates the task.

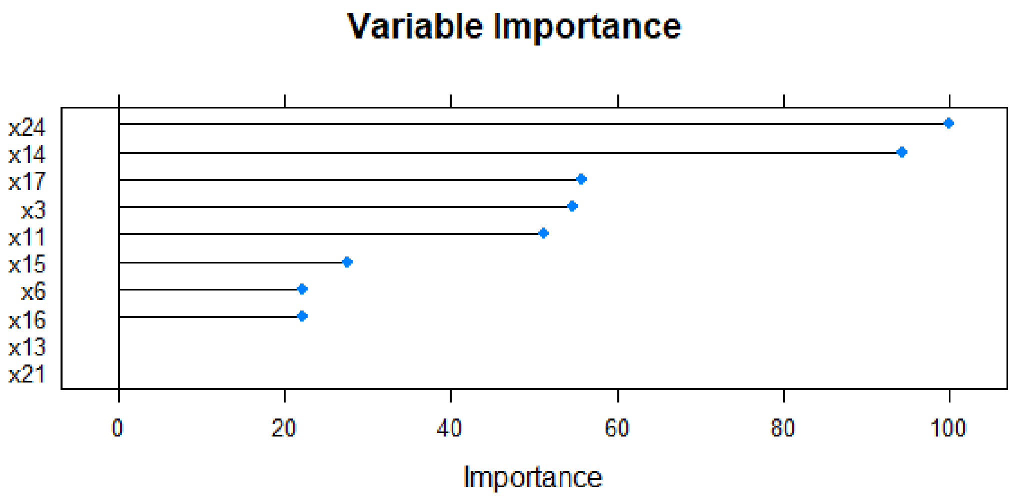
We used `corrplot` package to graphical display a correlation matrix. Positive correlations are displayed in blue and negative correlations in red colour. Colour intensity and the size of the circle are proportional to the correlation coefficients. Every correlation coefficient ranges from 1 to -1, where 1 is a prefect positive correlation, -1 a perfect negative correlation, and 0 indicates no correlation between two features.

Figure 3: Correlation Matrix



The `caret`-package includes a command to perform variable importance. `The varImp()` function calculates important features of almost all models (Kuhn, et al., 2017). We use a call to `plot` to visualize the results for importance scores generated by the `varImp()` function. We apply the `top=10` option to make the chart more readable and select the ten most important variables.

Figure 4: Variable Importance - varImp() function



We can see that the `varImp function` considers Accounts Payable Turnover (`x24`) to be the most important feature to predict German default firms, followed by Interest Coverage Ratio Leverage (`x14`), Quick Ratio `(x17)`, Operating Income/Total Assets `(x3)` and Net Indebtedness Leverage `(x11).`

All in all, decision trees are an attractive method since they work with many types of data, be it numerical or categorical, without any distributional assumptions and without pre-processing. Furthermore, most of `R`'s implementations handle missing data. Additionally, this method is robust to redundant and nonlinear data. The algorithm is easy to use, and the graphical output is relatively easy interpretable. On the other hand, decision trees have some shortcomings. They tend to overfit, especially without pruning, because the decision tree can `memorize` the training set. The model then becomes very specific to the training data and consequently has less ability to classify the unobserved data correctly. Besides, they tend to have a high training variance, which means that samples drawn from the same population can produce trees with different structures and different prediction accuracy. Additionally, a prediction accuracy can be eventually low, compared to other methods. For these reasons a technique called bagging is often used to improve decision tree models. Random Forests are a more specialized approach, which directly combines decision trees with bagging and yields remarkably better results (Zumel & Mount, 2014).

7 Random Forest

The Random Forest approach often gives an improved predictive accuracy for relatively complex tree models. This ensemble method combines several individual classification trees in the following way: from the original sample several bootstrap samples are drawn, and an unpruned classification tree is fitted to each bootstrap sample. The variable selection for each split in the classification tree is conducted only from a small random subset of predictor variables. From the complete forest the status of the response variable is predicted as an average or majority vote of the predictions of all trees (Maindonald & Braun, 2010).

Listing 20: |**random_forest_modell_fin.R**|

```r
mod_forest_I=randomForest(as.factor(status)~.,
                           data = training_complete,
                           importance = T,
                           ntree = 2000,
                           maxnodes = 100,
                           norm.votes = F)
```

We pass several arguments in to the `randomForest()` function. First of all, since it is a classification problem we define our target variable `status` as categorical by using `as.factor()` (Stephens, 2016). In comparison to CART models, trees are grown independently to their maximum extent, limited however by `nodesize` (minimum number of trees at a node). This parameter implicitly sets the depth of generated trees. Additionally, `maxnodes` can be used to limit the number of nodes (Maindonald & Braun, 2010).

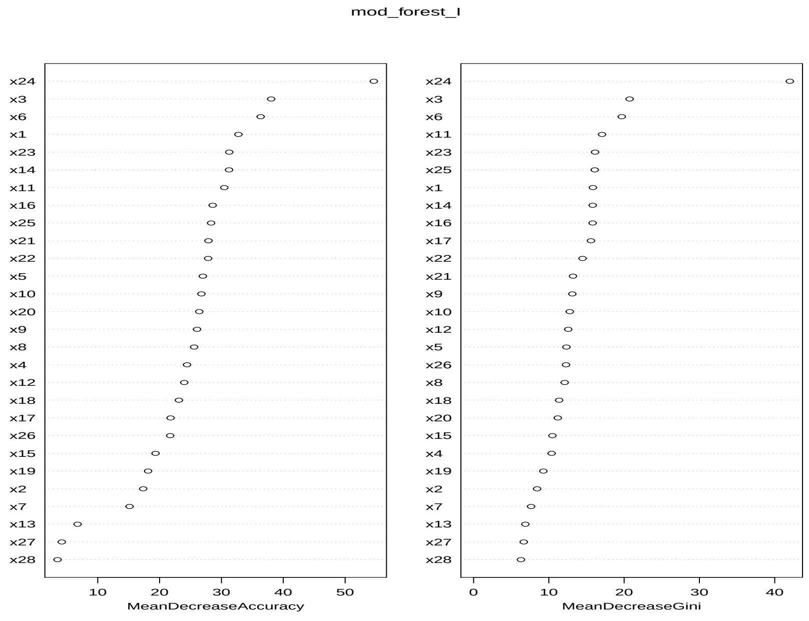
The main tuning parameter is the number `mtry` of variables that are randomly sampled at each split. By default, the `randomForest()` function in `R` draws square root of the total number of variables at each node for classification trees. Essentially, `mtry` controls the trade-off between the amount of information in each individual tree, and the correlation between trees. Smaller `mtry` values will grow the trees faster. However, in our case using a larger `mtry` is better, since we have a large number of variables to choose from, of which only a small fraction is actually useful. By doing so, we are more likely to draw some useful variables at every step of the tree-growing procedure (Zumel & Mount, 2014).

Further, the `ntree` argument specifies how many trees we want to grow. By tuning this parameter, the final accuracy of a model might increase up to some point. Also, limiting the number of trees restricts the complexity and computing time (Maindonald & Braun, 2010).

Random forests highly increase the prediction accuracy compared to individual classification trees. However, the interpretability of a random forest is not as straightforward as that of an individual classification tree, where the influence of a predictor variable directly corresponds to its position in the tree. Thus, alternative measures for variable importance are required for the interpretation of random forests (Strobl, et al., 2007).

We can calculate the variable importance by specifying `importance¯TRUE` in the `randomForest()` function, and then calling the functions `importance()` and `varImpPlot()`. Two measures of importance for each variable are then calculated: `MeanDecreaseAccuracy` and `MeanDecreaseGini`. The first measure is a `leave-one-out` type assessment of variable's contribution to prediction accuracy. Thereby the variable's values are randomly permuted in the out-of-bag samples, and the corresponding decrease in each tree's accuracy is estimated. If the average decrease over all the trees is large, then the variable is considered important - its value makes a big difference in predicting the outcome. If the average decrease is small, then the variable does not make much difference to the outcome. The second measure shows how each variable affects the quality of the tree by representing the decrease in node purity that occurs from splitting on a permuted variable (Maindonald & Braun, 2010).

Figure 5: Variable Importance - `importance()`-function



Both measures rate equally the first three and the fifth variables: Accounts Payable Turnover `x24`, Operating Income/Total Assets `x3`, EBITDA `x6` and Account Receivable Turnover Activity `x23`. Interestingly, this ranking order of `importance` differs from the results obtained with `varImp()` function applied for the CART model. Knowing which variables are most important or at least, which variables contribute the most to the structure of the underlying decision trees, can help us with variable reduction. This is useful not only for building smaller, faster trees, but for choosing variables to be used by another modelling algorithm, should that be desired.

The `random forest` package is robust and very user friendly. However, it is important to note that it has a drawback in that it is invariably biased towards features with many cut points. There is a package `party` which provides the `cforest()` command to build a forest of conditional inference trees. The basic construction of each tree is fairly similar to a Random Forest. Hence the decisions are made in slightly different ways, using a statistical test rather than a purity measure. The `party` package is better than the `randomForest` package in terms of accuracy, since `cforest` uses the weighted average of the trees to get the final ensemble. However, it is computationally more expensive (Modi, 2016). In our dataset we do not have features which have many categories and therefore we favour the `random forest` package to calculate the insolvency predictions. Additionally, it is computationally more efficient.

8 Evaluation of predictions

In this section we explain the steps followed in the `evaluate_predictions` quantlet. Purpose of this quantlet is to create a function that will take `labels` (actual solvency status) and `predictions` about the solvency status of firms and will return the confusion matrix, the ROC curve and the AUC as well as some evaluation metrics like sensitivity, specificity, precision and accuracy.

The confusion matrix, also known as error matrix, is a matrix that illustrates the performance of a classification model on a set of test data with known true values (labels). Each row of the matrix corresponds to a case of a predicted class while each column corresponds to a case of an actual class. It is called confusion matrix because it depicts whether the classifier is confusing the two classes, i.e. if it is wrongly labelling one class as another. It is a special case of a contingency table, with two dimensions ("actual" and "predicted") and two "classes" (solvent and insolvent in our case) in each dimension. The cells of our confusion matrix will present the number of:

1. True Positives (TP): firms who are correctly predicted to be insolvent (hits).
2. False Positives (FP): solvent firms were wrongly predicted to be insolvent (false alarm or Type I error)
3. False Negatives (FN): insolvent firms were wrongly predicted to be solvent (mass or Type II error)
4. True Negatives (TN): firms were correctly predicted to be solvent (correct rejection)

These values will then be used to get some important metrics which are described next.

Sensitivity and specificity are assessment metrics of the discriminative power of classification methods (Härdle et al., 2012). Sensitivity (also known as the true positive rate) measures the proportion of positives (defaulted firms in our case) that are correctly identified as such (Fawcett, 2006). It is defined as

$$TPR = \frac{TP}{TP + FN}.$$

Specificity (or true negative rate) measures the proportion of negatives (solvent firms in our case) that are correctly identified as such (Fawcett, 2006). It is defined as

$$TNR = \frac{TN}{TN + FP}.$$

Precision is analogous to the positive predictive value (PPV) and is a measure of exactness (Härdle et al., 2012). It is defined as

$$PPV = \frac{TP}{TP + FP}.$$

Finally accuracy measures the fraction of correct predictions and is defined as

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}.$$

The values of specificity and sensitivity will allow us to plot the ROC curve. The ROC curve (Receiver Operating Characteristic curve) is a graphical plot which depicts the diagnostic ability of a binary classification model for different values of its discrimination threshold. In our case, this threshold refers to the probability value above of which a firm is predicted to be insolvent. It is created by plotting the sensitivity values(y axis) against their corresponding $1 - specifity$ values (x axis) as the threshold varies (Fawcett, 2006). The area under the ROC curve (AUC) can be interpreted as the average power of the test on default or non-default corresponding to all discrimination thresholds (Härdle et al.(2012)). A larger AUC corresponds to a better classification result. A model with perfect discriminative power will have an AUC value of 1, while a random model without discriminative power will have an AUC value of 0.5, i.e. its ROC curve will be the 45 degree line. Thus, any reasonable rating model is expected to have an AUC value above 0.5, while AUC values close to 1 will indicate models with high diagnostic ability.

8.1 Implementation

Listing 21: |**Evaluate_Predictions.fin.R**|

```
19 get_prediction = function(fitted_probs, threshold){
20   predictions = ifelse(fitted_probs > threshold, 1, 0)
21   return(predictions)
```

The first function we created in this quantlet is the `get_prediction()`-function (lines 19-22). This function takes as inputs the fitted probabilities for insolvency and the threshold above of which a firm would be predicted to be defaulted. Using the `ifelse()` function in the body of this function, `get_prediction()` will return predictions for the solvency status of each firm. We construct then the `evaluate_predictions()` function (lines 24-47). Inputs of this function are the actual solvency status of each company (labels), the corresponding predictions about the status and the `verbose` parameter which is equal to `FALSE` by default. In the body of the function we create first the previously mentioned confusion matrix which we expect to be a 2x2 matrix. If it is not the case, the test in line 30 will show us a warning message. Then we get the values of TP, TN, FP and FN and we compute the values of sensitivity, specificity, precision and accuracy, which are then saved as a list in `reports`. The function will also print a data frame with those reports if we change the logical value of `verbose` to `TRUE`.

Listing 22: |**Evaluate_Predictions.fin.R**|

```
24 evaluate_predictions = function(labels, predictions, verbose = FALSE){
25   ct = table(factor(x = labels, levels = c(0, 1)),
26              factor(x = predictions, levels = c(0,1)))
27
28   if(any(dim(ct) != 2)) stop("Labels or Predictions contain more than 2 classes or
       both consist of only one class.")
29
30   # Unpack values from table (for easier reference):
31   TN = ct[1, 1]
32   TP = ct[2, 2]
33   FP = ct[1, 2]
34   FN = ct[2, 1]
35
36   # Calculate measures
37   reports = list(
38     sensitivity = TP / (TP + FN),
39     specificity = TN / (FP + TN),
40     precision = TP / (TP + FP),
41     accuracy = (TP + TN) / sum(ct))
42
43   if(verbose) print(data.frame(reports), digits = 3)
44   return(reports)
45 }
```

The final function created in this quantlet is the `evaluate_model()` function (lines 49-101), which takes the fitted probabilities for bankrupcy and the actual status (labels) as inputs. At first we create a threshold list in the body of the function with threshold values starting from 0 to 1, which is increased by miniscule steps. Then we apply the `get_prediction()` function to this list in order to get a list of predictions for each threshold value which we store as `pred_list`. Afterwards we apply the `evaluate_predictions()` function to `pred_list` and we get a list of reports. We use then the values of sensitivity and specificity from `reports` in order to compute the ROC curve and the area under it (AUC) (lines 58-70). For calculating the AUC we create the function `get_auc()` which takes the values of $1 -$ `specificities` and `sensitivities` as inputs. This function uses the trapezoidal method to approximate the AUC. For each difference in $1 - specificities$ there is one rectangle which underestimates the area under curve (corresponding to the "left" value of sensitivities) and one rectangle that overestimates it (corresponding to the "right" value of sensitivities). Therefore the function approximates the AUC by taking the average of the two rectangles.

Listing 23: |**Evaluate_Predictions.fin.R**|

```
49  pred_list = lapply(threshold_list,
50                     get_prediction,
51                     fitted_probs = fitted_probs)
52  report_list = lapply(pred_list,
53                       evaluate_predictions,
54                       labels = labels)
55  reports = unlist(report_list)
56  # Calculate ROC:
57  sensitivities = reports[names(reports) == "sensitivity"]
58  specificities = reports[names(reports) == "specificity"]
59  # Calculate AUC:
60  get_auc = function(x, y){
61    abs(sum(diff(x) * (head(y, -1) + tail(y, -1)))/2)}
62  auc = get_auc(1-specificities, sensitivities)
63  # Choose optimal metrics:
64  opt_ind = which.max(sensitivities + specificities - 1)
65  opt_sensitivity = sensitivities[opt_ind]
66  opt_specificity = specificities[opt_ind]
67  opt_threshold = seq(from = 0, to = 1, by = 0.0001)[opt_ind]
68  opt_predictions = get_prediction(fitted_probs = fitted_probs,
69                                   threshold = opt_threshold)
70  opt_accuracy = (ct[1, 1] + ct[2, 2]) / sum(ct)
71  opt_precision = precision = ct[2, 2] / (ct[2, 2] + ct[1, 2])
72  ct = table(factor(x = labels, levels = c(0, 1)),
73             factor(x = opt_predictions, levels = c(0,1)))
74  plot(x = 1 - specificities, y = sensitivities,
75       main = "ROC-Curve",
76       xlab = "False Positive Rate (1 - specificity)",
77       ylab = "True Positive Rate (sensitivity)",
78       xlim = c(0, 1),
79       ylim = c(0, 1),
80       asp = TRUE,
81       type = "s")
82  abline(c(0, 0), c(1,1), col = "grey")
83  return(list(sensitivity = opt_sensitivity,
84              specificity = opt_specificity,
85              accuracy = opt_accuracy,
86              precision = opt_precision,
87              threshold = opt_threshold,
88              predictions = opt_predictions,
89              auc = auc,
90              contingency_table = ct))}
```

We are next interested in finding the values of sensitivities and specificities for which the distance between the ROC curve and the 45-degree line (i.e. the ROC curve of a model with no discriminative power) is maximized. We use the index of these values in order to find the optimal values of sensitivity, specificity and the threshold, which is then used to compute the optimal predictions. These predictions are then used to create the optimal confusion matrix and get the optimal accuracy (lines 73-84). In the final part of the `evaluate_model()` function the ROC curve is plotted and the function returns a list with the optimal measures calculated above.

9 Results and Conclusions

Logistic regression and decision trees are benchmarking methods in predicting company credit risk. Their modelling techniques are well known for being able to return good probability estimates. Such models can be evaluated in terms of their estimated probabilities.

Most of the performance measures of a classifier can be read off the entries of a confusion matrix. Accuracy measure is largely used to measure a prediction power of a model. Besides, we seek to obtain an idea of more detailed measures such as sensitivity, specificity, and precision. Additionally, we produce the receiver operating characteristic curve (ROC curve) for each method to compare their discriminative power. This metric might provide interesting information for the investors, whose ultimate goal is to reduce losses due to company's bankruptcy.

The evidence from empirical results consistently shows that a credit risk model based on random forest significantly outperforms CART and Logit approach in modelling the default risk of German firms. Logit model and CART show almost equal accuracy rates, though the values of the area under the curve (AUC), show that the CART classifier does not perform better than a random guess.

In summary, we successfully managed to replicate most of the results reported in the papers. Additionally we implemented custom functions for censoring a dataset by replacing extreme values by the 0.05 and 0.95-percentile and for computing Linear Discriminant Analysis. Additionally we implemented *logit* and *probit* models, followed by an application of the *logit*-model to the creditreform dataset, although we didn't use our own implementation for reasons of efficiency. We applied Classification and Regression Trees as well as Random Forests and evaluated their predictions using our own function for calculating Accuracy, Sensitivity, Specificity, Precision measures and plotting the Receiver Operator Characteristic as well as calculating the Area Under Curve.

Figure 6: ROC Logit

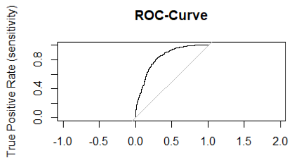


Figure 7: ROC CART

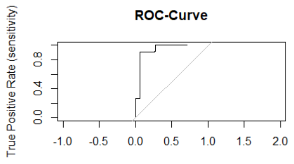


Figure 8: ROC RF

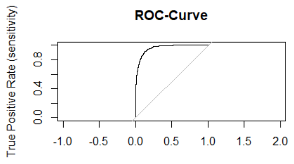


Table 6: Classifier performance measure

Performance measure	Logit (%)	CART(%)	Random forest (%)
Accuracy Rate	71,6	74,6	87,2
Sensitivity	81,6	90,6	91,2
Specificity	70,9	73,5	87,0
Precision	15,5	18,3	31,4
AUC	82,9	56,1	95,8

10 Appendix

10.1 Unit Tests LDA

In order to test the `lda`-function we take a dataset from which we know that it is linearly separable already and see if we can reproduce the results. Since Linear Discriminant Analysis was first proposed by Sir Ronald A. Fisher it is only fitting that we test it on his famous `iris` dataset. The `iris` dataset contains measurements of 150 observations for the sepal and petal length and width of three different species of iris flowers. We know that the sepal length and the petal length are sufficient variables to create an obviously visible separation line in figure 10.1.1 which discriminates the Setosa-species almost perfectly from the rest. An exception is one observation of the setosa speces which seems to be an outlier.

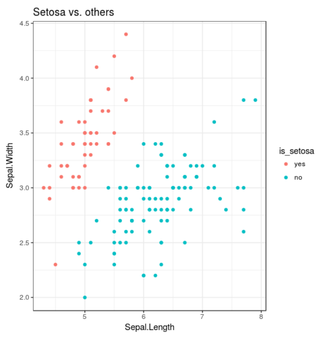


Figure 9: Setosa species are already linearly separable.

10.1.1 Expectations

Should the `lda`-function work as intended we would expect this result only to become better, i.e. the points being perfectly separable. We first import the `iris` dataset and transform the Species column in order to fit our binary task, that is we create a dichotomous variable indicating if the particular flower is of the Setosa species or not. We can then directly use the `lda`-function and plot the first two components against each other.

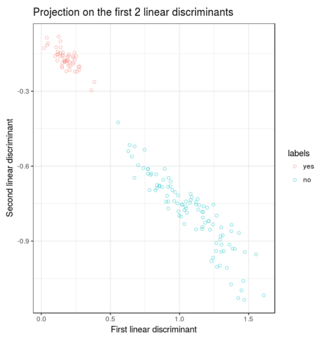


Figure 10: LDA manages to separate both groups.

10.1.2 Test

Listing 24: |**lda_tests.R**|

```r
irisLDA = lda(iris_test, "is_setosa")
plot(irisLDA)
```

10.1.3 Result

As we can see in figure 10.1.2 the two classes are perfectly separable. In fact they would even be separable using only the first linear component.

10.2 Unit Tests Gradient Descent

In order to test the `gradientDescentMinimizer()`-routine we will try it out on a function that is easy to optimize analytically, but tricky when it comes to numerical methods using finite differences. We opt for the function

$$f(x) = 0.008 \cdot x^4 + 2.$$

10.2.1 Expectations

Of course the first derivative of f is equal to

$$f\prime(x) = 0.032 \cdot x^3.$$

Which, set to zero, yields a solution at $x^\star = 0$. Obtaining this result is easy using calculus, however the `gradientDescentMinimizer()` computes its approximate gradients using only finite differences. The derivative of function f tends towards 0 pretty early making it hard for an algoritmic procedure to decide how close to 0 is going to be close enough. This may result in an early abort of the update steps which yields unprecise results.

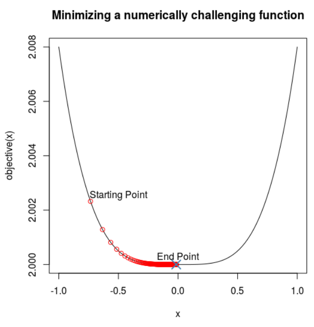


Figure 11: Although the gradient decays very fast, it is possible to reach the exact minimum.

10.2.2 Test

Listing 25: |unit_tests.R|

```r
source("BRM/utils.R")
set.seed(1234)

# Test univariate Minimizer:
objective = function(x){
  return(0.008*x^4 + 2)
}
curve(objective(x), xlim=c(-1, 1), main = "Minimizing a numerically challenging
  function")
res = univariateMinimizer(obj = objective,
                                max_iter = 1000000,
                                precision = 1e-7,
                                learn_rate = 0.1,
                                verbose = TRUE)

points(res, objective(res), col = "steelblue", pch=4, lwd=2, cex=2)
text(-0.5, y=objective(-0.75), labels="Starting Point")
text(0, 2.0003, labels="End Point")
```

10.2.3 Result

We see that the `gradientDescentMinimizer()` finds the correct minimum. However, this result has been obtained only after rigorous tuning. In itself this is not a grave shortcoming since it is inherent to any gradient-Descent-routine. All in all we would recommend running the routine multiple times to ensure that it converged towards the same minimum.

10.3 Unit Tests Evaluation

In order to test the functions used for evaluating the predictions we compare our custom function with the `auc()` and `plot.roc()`-functions from the `pROC`-package. We load the `Boston` dataset from the `MASS`-package, which contains data from 506 different census districts of the Boston metropolitan area.

10.3.1 Expectations

For testing purposes we will build a logit-model trying to predict wether the census district is adjacent to the Charles River which is stored in the variable `chas`. These models are of course nonsense. We will calculate the AUC and plot the ROC-curve once with our home-made function and compare them with the functions from the pROC-package. We would expect that the absolute difference between the two AUC-measures should not be larger than 0.001.

10.3.2 Test

Listing 26: |**unit_tests.fin.R**|

```r
## Unit tests & assorted things
# Test the evaluation function using a logit model of the Boston-Housing dataset.

source("Evaluation/Evaluate_Predictions.fin.R")

library("pROC")
library("MASS")

data("Boston")

test_eval_fun = function(formula){
  model = glm(data = Boston, formula = formula, family = binomial(link = 'logit'))
  par(mfrow = c(1,2))
  custom_auc = evaluate_model(model$fitted.values, Boston$chas)
  plot.roc(Boston$chas, model$fitted.values)
  r_auc = auc(Boston$chas, model$fitted.values)

  par(mfrow = c(1,1))
  # Is the difference between AUCs large?
  test = abs(r_auc - custom_auc$auc) <= 0.001
  return(test)
}

# Take every single variable as predictor for the Charles-River-Index (except for
  chas).
models = paste("chas ~", names(Boston)[-4])
tests = sapply(models, test_eval_fun)
print(all(tests))

if(!all(tests)) {
  print(tests)
  sum(tests) / length(tests)}
```

10.3.3 Result

The function passes eleven out of the thirteen tests (84 %). The plotted ROC-curves were remarkably similar, and the failed tests were caused by especially bad models yielding almost no predictive power. The differences can be mostly attributed to a design decision, we used a step-function, whereas the authors of the pROC-package use a line-plot. However we consider this test a success.

10.4 Code

10.4.1 Preparation

Listing 27: |**data.preparation.R**|

```r
library("dplyr")

#import data
data= read.csv("Data/SPL_data.csv", sep = ";", dec = '.', header = TRUE,
               stringsAsFactors = TRUE)

# Due to missing insolvencies in 1996 and missing data from 2003 onwards,
# we choose only the data of the period 1997-2002
#data1 = data 1997-2002
data1 = filter(data, JAHR>= 1997 & JAHR<=2002)

# Extract the industry class of companies
data1$Ind.Klasse = substring(data1$VAR26, 1, 2)

# As we are only interested in companies with high percentage in the industry
# composition we choose only companies belonging to the following sectors
# (according to German Classification of Economic Activities Standards (WZ 1993)):
#     1. Manufacturing (Man)
#     2. Wholesale and Retail (WaR)
#     3. Construction (Con)
#     4. Real Estate (RE)
Man = filter(data1, Ind.Klasse %in% as.character(15:37))
Man$Ind.Klasse = "Man"

WaR = filter(data1, Ind.Klasse %in% as.character (50:52))
WaR$Ind.Klasse = "WaR"

Con = filter(data1, Ind.Klasse == "45")
Con$Ind.Klasse = "Con"

RE = filter(data1, Ind.Klasse %in% as.character(70:74))
RE$Ind.Klasse = "RE"

# Remove data and data1 & bind the above subsets to get one dataset containing
# only companies of interest

rm(data, data1)

data = rbind(Man, WaR, Con, RE)

# Furthermore we choose only companies whose total assets (VAR6) are in the range
# 10^5 - 10^8

data = data[data$VAR6 >= 10^5 & data$VAR6<= 10^8,]

############################################################################
# Eliminate observations with 0 value for the following variables used as
  denominators
# in calculation of financial ratios to be used in classification:

# total assets (VAR6)
# total sales(VAR16)
# cash (VAR1)
# inventories (VAR2)
# current liabilities (VAR12)
# total liabilities (VAR12 + VAR13)
# total assets-intangibe assets-cash-lands and buildings (VAR6 - VAR5 - VAR1 - VAR8
  )
# interest expenses (VAR19)


data_clean = data %>% filter(VAR6 != 0,
                             VAR16 != 0,
                             VAR1 != 0,
                             VAR2 != 0,
                             VAR12 != 0,
                             VAR12 + VAR13 != 0,
                            VAR6 - VAR5 - VAR1 - VAR8 != 0,
                            VAR19 != 0)

# Get insolvent firms:
insolvent = data_clean[data_clean$T2 == 1, "ID"]

# Get solvent firms:
solvent = data_clean[data_clean$T2 == 0, "ID"]

#Show table with number of solvent/insolvent firms
table(data_clean$JAHR, data_clean$T2)

# Define Function that overwrites outliers:
# '...' : anonymous arguments that get passed to the quantile function.
replace_extreme_values = function(x, ...){
  # The point of this is to allow more flexibility for the user.
  quantiles = quantile(x, ...)

  x = ifelse(x <= quantiles[1], quantiles[1], x)
  x = ifelse(x >= quantiles[2], quantiles[2], x)

  return(x)
}


# add columns with financial ratios for each firm to the dataset
test_data = data_clean %>%
  mutate(x1 = VAR22/VAR6,
         x2 = VAR22/VAR16,
         x3 = VAR21/VAR6,
         x4 = VAR21/VAR16,
         x5 = VAR20/VAR6,
         x6 = (VAR20+VAR18)/VAR6,
         x7 = VAR20/VAR16,
         x8 = VAR9/VAR6,
         x9 = (VAR9-VAR5)/(VAR6-VAR5-VAR1-VAR8),
         x10 = VAR12/VAR6,
         x11 = (VAR12-VAR1)/VAR6,
         x12 = (VAR12+VAR13)/VAR6,
         x13 = VAR14/VAR6,
         x14 = VAR20/VAR19,
         x15 = VAR1/VAR6,
         x16 = VAR1/VAR12,
         x17 = (VAR3-VAR2)/VAR12,
         x18 = VAR3/VAR12,
         x19 = (VAR3-VAR12)/VAR6,
         x20 = VAR12/(VAR12+VAR13),
         x21 = VAR6/VAR16,
         x22 = VAR2/VAR16,
         x23 = VAR7/VAR16,
         x24 = VAR15/VAR16,
          x25 = log(VAR6),
          x26 = VAR23/VAR2,
          x27 = VAR24/(VAR12+VAR13),
          x28 = VAR25/VAR1)

# Prepare data frame containing relative variables:
#   ID, T2, JAHR and financial ratios to be used for classification

test_data_rel = select(test_data,
                       ID, T2, JAHR, x1:x28)

# Prepare subsets of solvent and insolvent companies
# and replace extreme values
test_data_insolvent = test_data_rel %>% filter(T2 == 1)
test_data_solvent = test_data_rel %>% filter(T2 == 0)

test_data_insolvent[, -c(1:3)] = apply(X = test_data_insolvent[, -c(1:3)],
                                       MARGIN = 2,
                                       FUN = replace_extreme_values,
                                       probs = c(0.05, 0.95))

test_data_solvent[, -c(1:3)] = apply(X = test_data_solvent[, -c(1:3)],
                                     MARGIN = 2,
                                     FUN = replace_extreme_values,
                                     probs = c(0.05, 0.95))

# Print descriptive statistics
print("Solvent:")
print(round(t(apply(X = test_data_solvent[, -c(1:3)],
              MARGIN = 2,
              quantile,
              probs = c(0.05, 0.5, 0.95))), digits = 2))

print("Insolvent:")
print(round(t(apply(X = test_data_insolvent[, -c(1:3)],
          MARGIN = 2,
          quantile,
          probs = c(0.05, 0.5, 0.95))), digits = 2))

data_clean = rbind(
  test_data_insolvent,
  test_data_solvent
)

# Remove unnecessary variables from working environment:
rm(list = ls()[!ls() == "data_clean"])


# Result: test_data_rel contains 9591 solvent and 783 insolvent firms and their fin
  .ratios
#        Almost same result as Zhang, Hardle (2010)
print(table(factor(data_clean$T2,
            levels = c(0, 1),
            labels = c("Solvent", "Insolvent"))))
```

10.4.2 BRM

Listing 28: |**utils.R**|

```r
# Define Optimization Routines:
```

```r
#      - univariate Case.
#      - bivariate Case.
#      - multivariate Case.

# Define Other helper-functions:
#      - get all numeric columns of a dataframe
#      - get mean vector, grouped by class label
#      - get covariance matrix, grouped by class label

# ===============================================================================
## Optimization Routines
# ===============================================================================
# Univariate Case:
univariateMinimizer = function(obj, learn_rate = 1, epsilon = 1e-3,
                               max_iter = 100, precision = 1e-6, verbose = FALSE,
                               report_freq = 100){

  # Approximate gradient by taking the difference of an epsilon-step
  d = function(x, fun = obj, e = epsilon){
    gradient = (fun(x + e) - fun(x)) / e
    # Sometimes the approximate gradient turns out to be too steep.
    # Thus we bound it which stabilizes the algorithm by introducing
    # some inefficiency.
    gradient = min(abs(gradient), 10) * sign(gradient) # Correct the sign
    return(gradient)
  }

  # Draw multiple random starting points.
  # Take the combination that provides the lowest value of the objective function.
  a = runif(1000, min = -1000, max = 1000)
  a = a[which.min(sapply(a, obj))]
  i = 0

  gradient = d(a)
  if(verbose){
    print("Starting Point:")
    print(paste("a:", a, "f(a):", obj(a), "df(a):", d(a)))
  }

  while (i < max_iter & abs(gradient) > precision) {
    i = i + 1

    gradient = d(a)
    a = a - learn_rate * gradient

    if(i %% report_freq == 0 & verbose) {
      cat("Iteration:", i, "x:", a, "f(x):", obj(a), "df(x):", d(a),
            sep = c("\t", "\n"))
      points(a, obj(a), col = "red")}
  }
  cat("\t", "Results:\n","Iteration:", i, "x         :", a, "f(x)      :", obj(a), "
    df(x)     :", d(a),
      sep = c("\t", "\n"))

  if(i >= max_iter){
    warning("Maximum number of iterations reached.")}
  return(a)
}

# ===============================================================================
# Bivariate Case:
bivariateMinimizer = function(obj, epsilon_step = 0.001, max_iter = 10,
                               precision = 1e-6, learn = 0.5, verbose = FALSE,
                               report_freq = 100){
  # Approximate Gradient
  get_2D_gradient = function(x, objective = obj, epsilon = epsilon_step){
    init = matrix(data = x, nrow = 2, ncol=2, byrow = TRUE)
    steps = init + diag(x = epsilon, ncol = 2, nrow = 2)

    f_steps = apply(steps, 2, objective)
    f_comp =  apply(init, 2, objective)

    D = (f_steps - f_comp) / epsilon
    # Trim D to limit the gradient:
    D_trimmed = ifelse(abs(D) <= 5, abs(D), 5) * sign(D)
    return(D_trimmed)
  }
  # Draw multiple random starting points.
  # Take the combination that provides the lowest value of the objective function.
  a = matrix(data = runif(10000, min = -100, max = 100),
             ncol = 2)
  # DEBUG:
  f_a = apply(a, 1, obj)
  a = a[which.min(f_a), ]
  gradient = get_2D_gradient(a)

  i = 0

  learn_rates = seq(from = learn, to = 0, length.out = max_iter + 1)

  while(i <= max_iter & any(abs(gradient) >= precision)){
    if(i %% report_freq == 0 & verbose) {
      cat("\nStep:\t\t", i,
          "\nx:\t\t", a,
          "\ngradient:\t", gradient,
          "\nlearn:\t", learn_rates[i],
          "\n---------")}

    i = i + 1
    a = a - learn_rates[i] * gradient
    gradient = get_2D_gradient(a)
  }

  cat("\nResults\n",
      "\nIteration:\t", i,
      "\nx:\t\t", a,
      "\nf(x):\t\t", obj(a),
      "\ndf(x):\t\t", gradient)

  if(i >= max_iter){
    warning("Maximum number of iterations reached.")}
  return(a)
}

# ==============================================================================
# Multivariate Case:
gradientDescentMinimizer = function(
  obj, n_pars, epsilon_step = 0.001,
  max_iter = 10, precision = 1e-6,
  learn = 0.5, verbose = FALSE,
  report_freq = 100){
  # Check for invalid parameters:
  stopifnot(epsilon_step > 0,
            max_iter > 0,
             precision > 0,
             learn > 0,
             report_freq > 0)

  # Approximate Gradient
  get_gradient = function(x, d = n_pars,
                          objective = obj,
                          epsilon = epsilon_step){
    init = matrix(data = x, nrow = d, ncol = d, byrow = TRUE)
    steps = init + diag(x = epsilon, ncol = d, nrow = d)
    f_steps = apply(steps, 1, objective)
    f_comp =  apply(init, 1, objective)
    D = (f_steps - f_comp) / epsilon
    D_trimmed = ifelse(abs(D) <= 100, abs(D), 100) * sign(D)
    return(D_trimmed)}

  # Draw multiple random starting points.
  # Use the one that provides the lowest value of the objective function.
  a = matrix(data = runif(1000 * n_pars,
                          min = -10,
                          max = 10),
             ncol = n_pars)
  f_a = apply(a, 1, obj)
  a = a[which.min(f_a), ]
  gradient = get_gradient(a)

  # Set up values for loop
  i = 0
  learn_rates = seq(from = learn, to = 0, length.out = max_iter + 1)

  while(any(abs(gradient) >= precision) & i <= max_iter){
    if(i %% report_freq == 0 & verbose) {
      cat("\nStep:\t\t", i,
          "\nx:\t\t", a,
          "\ngradient:\t", gradient,
          "\nlearn:\t", learn_rates[i],
          "\n------------------------------------")}

    i = i + 1
    a = a - learn_rates[i] * gradient
    gradient = get_gradient(a)
  }

  cat("\nResults\n",
      "\nIteration:\t", i,
      "\nx:\t\t", a,
      "\nf(x):\t\t", obj(a),
      "\ndf(x):\t\t", gradient,
      "\n")

  if(i >= max_iter){
    warning("Maximum number of iterations reached.")}
  return(a)
}

# ==========================================================================
## Other helpers
# ==========================================================================

get_numeric_cols = function(data){
  numerics = sapply(data, is.numeric)
  return(numerics)
```

```r
}

get_group = function(data, by, group){
  num = get_numeric_cols(data)
  x = subset(data, data[, by] == group,
             select = num)
  return(x)
}

get_class_means = function(Data, By, ...){
  groups = unique(Data[, By])
  grp = as.list(groups)
  sub_data = lapply(X = grp, FUN = get_group,
                    by = By, data = Data)
  # Get list of means.
  means = sapply(X = sub_data, FUN = colMeans, ...)
  colnames(means) = as.character(groups)
  return(means)
}

get_class_cov = function(Data, By, ...){
  groups = unique(Data[, By])
  grp = as.list(groups)
  sub_data = lapply(X = grp, FUN = get_group,
                    by = By, data = Data)
  # Get list of covariance matrices.
  covs = lapply(X = sub_data, FUN = cov, ...)
  names(covs) = as.character(groups)
  return(covs)
}
```

Listing 29: |**logit.R**|

```r
source("BRM/utils.R")

# Define a function that returns a log-likelihood function.
get_loglikelihood = function(data, y, distr){
  grp = unique(data[, y])
  nums = get_numeric_cols(data)

  Xy_mat = as.matrix(cbind(
    data[, which(nums)],
    y = ifelse(data[, y] == grp[1], 1, 0)))

  # y is always the last column
  y_pos = dim(Xy_mat)[2]

  l = function(x){
    - sum(apply(Xy_mat, 1,
                function(X){
                  X[y_pos] * distr(t(X[-y_pos]) %*% x,
                                   lower.tail = TRUE,
                                   log.p = TRUE) +
                    (1-X[y_pos]) * distr(t(X[-y_pos]) %*% x,
                                         lower.tail = FALSE,
                                         log.p = TRUE)}))}
  # Return loglikelihood as a function of x (here 'x' stands for the weights)
  return(l)
}

# Get logit-weights:
brm = function(data, y, mode, ...){
  distr = switch (mode,
                  "logit" = plogis,
                  "probit" = pnorm
  )

  # Add column of ones for the intercept:
  Data = cbind(constant = 1,
               data)

  llog = get_loglikelihood(Data, y, distr)
  params = get_numeric_cols(Data)

  # We use -obj because we want to maximize obj (i.e. the likelihood).
  beta = gradientDescentMinimizer(obj = llog,
                                  sum(params),
                                  ...)
  names(beta) = names(which(params))
  # Get fitted probabilities:
  Xb = as.matrix(Data[, params]) %*% beta
  Data$fitted_values = sapply(X = Xb, FUN = distr)
  Data$true_values = Data[, y]

  out = list(X = Data[, params],
             weights = beta,
             likelihood_val = llog(beta),
             type = mode,
             distribution = distr)

  return(structure(out, class = "brm"))
}

predict.brm = function(model, data){
  Xb = as.matrix(cbind(constant = 1, data)) %*% model$weights
  predictions = sapply(X = Xb,
                       FUN = model$distribution,
                       log.p = FALSE,
                       lower.tail = TRUE)
  return(predictions)
}
```

10.4.3 Logit

Listing 30: |**logit_model_fin.R**|

```
#==============================================================
## Building and applying logistic regression model for insolvecy
##
## steps to be taken:
## split dataset into training and test set
## apply bootstrapping technique and re-balance the training set by downsampling
   solvent firms
## fit the model and evaluate the outcomes
## build a confusion matrix and store the predictions for further analysis
#========================================================

source("Preparation/data.preparation.R")

library("caret")

### build training and test set:
```

```r
data_clean$T2 = factor(data_clean$T2, levels = c(0,1), labels = c(0,1))

training=subset(data_clean,data_clean$JAHR<2000)
validierung=subset(data_clean,data_clean$JAHR>=2000)

validierung_logit=validierung
names(validierung_logit)[2]="status"
validierung_logit=validierung_logit[,c(-1,-3)]

#Bootstrapping:

training_insolvent=training[training$T2==1,-c(1,3)]
training_solvent_full=training[training$T2==0, -c(1,3)]

TN=0
TP=0
FP=0
FN=0
set.seed(1234) #pseudo-random number generation to make work reproducible and
  results comparable
for (i in 1:30){
  randum_numb=round(runif(nrow(training_insolvent), min = 1, max = nrow(training[
    training$T2==0,])))
  training_solvent=training_solvent_full[randum_numb,]
  training_complete=rbind(training_insolvent,training_solvent)

  #fit the model:
  names(training_complete)[1]="status"
  glm_mod=train(as.factor(status)~.,data=training_complete, method="glm", family="
    binomial")
  # define target variable as categorical
  # specify distribution function and link function
  glm_pred = predict(glm_mod, newdata=validierung_logit)
  conf_mat_logit=table(glm_pred, validierung_logit$status)
  TN=TN+conf_mat_logit[1,1]
  FP=FP+conf_mat_logit[1,2]
  TP=TP+conf_mat_logit[2,2]
  FN=FN+conf_mat_logit[2,1]
}
# evaluate the model:
pred_logit=predict(glm_mod, newdata=validierung_logit,type="prob")
write.csv(cbind(labels=validierung_logit$status,pred_logit),file="Logit/logit_pred.
  csv")
```

10.4.4 CART

Listing 31: |**cart_modell_fin.R**|

```r
#===============================================================
## Building a decision tree
##
## steps to be taken:
## split dataset into training and test set
## apply bootstrapping technique and re-balance the training set by downsampling
  solvent firms
## fit the model and evaluate the outcomes
## calculate correlation matrix
## calculate variable importance and plot the results
## visualize classification tree
## build a confusion matrix and store the predictions for further analysis
```

```r
#===================================================================

source("Preparation/data.preparation.R")

library("caret")
library("corrplot")
library("rpart")
library("rpart.plot")
library("rattle")


### build training and test set:
data_clean$T2 = factor(data_clean$T2, levels = c(0,1), labels = c(0,1))

training=subset(data_clean,data_clean$JAHR<2000)
validierung=subset(data_clean,data_clean$JAHR>=2000)

validierung_cart=validierung
names(validierung_cart)[2]="status"
validierung_cart=validierung_cart[,c(-1,-3)]

#Bootstrapping:
training_insolvent=training[training$T2==1,-c(1,3)]
training_solvent_full=training[training$T2==0, -c(1,3)]

TN=0
TP=0
FP=0
FN=0
set.seed(1234) #pseudo-random number generation to make work reproducible and
  results comparable
for (i in 1:30){
  randum_numb=round(runif(nrow(training_insolvent), min = 1, max = nrow(training[
    training$T2==0,])))
  training_solvent=training_solvent_full[randum_numb, ]
  training_complete=rbind(training_insolvent,training_solvent)

  #fit the model:
  names(training_complete)[1]="status"
  modfit=train(status~.,method="rpart",data=training_complete)

  #calculate predictions and store the results in confusion matrix:
  pred.cart=predict(modfit,newdata=validierung_cart)
  conf_mat_cart=table(pred.cart,validierung_cart$status)
  TN=TN+conf_mat_cart[1,1]
  FP=FP+conf_mat_cart[1,2]
  TP=TP+conf_mat_cart[2,2]
  FN=FN+conf_mat_cart[2,1]
}
#calculate correlation matrix
correlations=round(cor(training_complete[, -1]),2)
pdf("CorrPlot.pdf")
corrplot(correlations, order = "hclust")
dev.off()
#perform variable importance
pdf("varImp.pdf")
plot(varImp(modfit), main="Variable Importance", top=10)
dev.off()

# use rpart for plotting a visually appealing decision tree and predictions
mod_fit=rpart(as.factor(status)~.,data=training_complete)

```

```r
pdf("fancyRpartPlot.pdf")
fancyRpartPlot(mod_fit)
dev.off()

pred_cart=predict(mod_fit,newdata=validierung_cart,type="prob")
write.csv(cbind(label=as.numeric(as.character(validierung_cart$status)),pred_cart),
  file="CART/cart_pred.csv")
```

10.4.5 Random_Forest

Listing 32: |random_forest_modell_fin.R|

```r
#=============================================================
## Building a random forest model (package randomForest)
##
## steps to be taken:
## split dataset into training and test set
## apply bootstrapping technique and re-balance the training set by downsampling
  solvent firms
## fit the model and evaluate the outcomes
## perform variable importance and plot the results
## build a confusion matrix and store the predictions for further analysis
#========================================================

source("Preparation/data.preparation.R")

library("randomForest")

### build training and test set:
data_clean$T2 = factor(data_clean$T2, levels = c(0,1), labels = c(0,1))

training=subset(data_clean,data_clean$JAHR<2000)
validierung=subset(data_clean,data_clean$JAHR>=2000)

validierung_rf=validierung
names(validierung_rf)[2]="status"
validierung_cart=validierung_rf[,c(-1,-3)]

#Bootstrapping:

training_insolvent=training[training$T2==1,-c(1,3)]
training_solvent_full=training[training$T2==0, -c(1,3)]

TN=0
TP=0
FP=0
FN=0

set.seed(1234)#pseudo-random number generation to make work reproducible and
  results comparable
for (i in 1:30){
  randum_numb=round(runif(nrow(training_insolvent), min = 1, max = nrow(training[
    training$T2==0,])))
  training_solvent=training_solvent_full[randum_numb,]
  training_complete=rbind(training_insolvent,training_solvent)

  #fit the model:
  names(training_complete)[1]="status"
  mod_forest_I=randomForest(as.factor(status)~.,
                            data = training_complete,
                              importance = T,
                              ntree = 2000,
                              maxnodes = 100,
                              norm.votes = F)
  # define target variable as categorical
  # specify number of trees to be grown
  # model predictions:
  pred_rf=as.data.frame(predict(mod_forest_I, validierung_rf))
  conf_mat_rf=table((as.numeric(unlist(pred_rf))-1),validierung_rf$status)
  TN=TN+conf_mat_rf[1,1]
  FP=FP+conf_mat_rf[1,2]
  TP=TP+conf_mat_rf[2,2]
  FN=FN+conf_mat_rf[2,1]
}
#perform variable importance
VarImp_rf=importance(mod_forest_I)#call importance on the bancrupcy model
                        #importance()function returns a matrix of
                        #importance measures (larger values=more important)
varImpPlot(mod_forest_I)#plot Var importance as measured by accuracy change
write.csv(imp,file="Random_Forest/importance.csv")

#calculate probabilities
pred_rf=as.data.frame(predict(mod_forest_I, validierung_rf, type = "prob"))
write.csv(cbind(label=validierung_rf$status,pred_rf),file="Random_Forest/rf_pred.
  csv")
```

10.4.6 LDA

Listing 33: |**utils.R**|

```
# Define helper-functions:
#     - get all numeric columns of a dataframe
#     - get mean vector, grouped by class label
#     - get covariance matrix, grouped by class label
# ==============================================================================

get_numeric_cols = function(data){
  numerics = sapply(data, is.numeric)
  return(numerics)
}

get_group = function(data, by, group){
  num = get_numeric_cols(data)
  x = subset(data, data[, by] == group,
             select = num)
  return(x)
}

get_class_means = function(Data, By, ...){
  groups = unique(Data[, By])
  grp = as.list(groups)
  sub_data = lapply(X = grp, FUN = get_group,
                    by = By, data = Data)
  # Get matrix of means.
  means = sapply(X = sub_data, FUN = colMeans, ...)
  colnames(means) = as.character(groups)
  return(means)
}

get_class_cov = function(Data, By, ...){
```

```r
  groups = unique(Data[, By])
  grp = as.list(groups)
  sub_data = lapply(X = grp, FUN = get_group,
                    by = By, data = Data)
  # Get list of covariance matrices.
  covs = lapply(X = sub_data, FUN = cov, ...)
  names(covs) = as.character(groups)
  return(covs)
}
```

Listing 34: |**utils.R**|

```r
library("geigen")

# Import helpers
source("LDA/utils.R")

lda = function(data, by){
  # Closed form solution of LDA:
  # w = S_b^-0.5 * largest_eigenvector(S_b^0.5 * S_w^-1 * S_b^0.5)

  # Check prerequisites
  num = get_numeric_cols(data = data)
  classes = unique(data[, by])

  stopifnot(
    length(num) > 0,
    length(classes) == 2)

  mu = get_class_means(Data = data, By = by, na.rm = TRUE)
  S = get_class_cov(Data = data, By = by, use = "complete.obs")
  n = sapply(classes, function(x){sum(data[, by] == x)})
  print(n)
  # Compute overall mean:
  x_bar = colMeans(data[, num], na.rm = TRUE)

  # Compute between class scatter matrix:
  Sb1 = (mu[, 1] - x_bar) %*% t(mu[, 1] - x_bar)
  Sb2 = (mu[, 2] - x_bar) %*% t(mu[, 2] - x_bar)
  S_b = Sb1 + Sb2

  # Compute within class scatter matrix:
  S_w = (n[1]-1)*S[[1]] + (n[2]-1)*S[[2]]

  # Extract eigenvector corresponding to largest eigenvalue
  # geigen solves the generalized eigenvalue problem of:
  # A*x = lambda B*x
  V = geigen(S_b, S_w, symmetric = TRUE)

  # Extract (absolutely) largest eigenvalue
  ev_order = order(abs(V[["values"]]), decreasing = TRUE)
  v = V[["vectors"]][ev_order[1:2], ]

  # Percent of variance explained:
  inertia = (V[["values"]][ev_order[1:2]])^2 / sum(V[["values"]]^2)

  # Compute rotations
  lda1 = apply(X = data[, num], MARGIN = 1, FUN = function(x){sum(x * v[1, ])})
  lda2 = apply(X = data[, num], MARGIN = 1, FUN = function(x){sum(x * v[2, ])})

  out = list(
    X = data.frame("lda1" = lda1,
                    "lda2" = lda2,
                    "labels" = data[, by]),
    classes = classes,
    class_means = mu,
    scalings = v,
    explained_var = inertia)

  return(structure(out, class = "flda"))
}

plot.flda = function(model){
  p = ggplot2::ggplot(data = model$X, ggplot2::aes_string(x = "lda1", y = "lda2",
    color = "labels")) +
    ggplot2::geom_point(alpha = 0.5, shape=21) +
    ggplot2::ggtitle("Projection on the first 2 linear discriminants") +
    ggplot2::xlab("First linear discriminant") +
    ggplot2::ylab("Second linear discriminant") +
    ggplot2::theme_bw()
  return(p)
}

predict.flda = function(model, data){
  v = model$scalings[1, ]
  num = get_numeric_cols(data)

  Data = data[, num]
  dims = dim(Data)
  stopifnot(dims[2] == length(v))

  # Compute discriminant as the dot product of every observation
  # with the scaling vector v:
  discr = apply(X = Data, MARGIN = 1, FUN = function(x){sum(x * v)})

  # Compute cutoff threshold:
  c = 0.5 * t(v) %*% (rowSums(model$class_means))
  predictions = ifelse(discr <= as.numeric(c), model$classes[1], model$classes[2])

  return(predictions)
}
```

10.4.7 Evaluation

Listing 35: |**Evaluate_Predictions.fin.R**|

```
#=================================================================
# Evaluate Predictions
#
## Intended Outcome:
## Function that takes labels and predictions as inputs
## and returns the following KPI's/Plots/Tables:
##     - Confusion Matrix
##         - true positives
##         - true negatives
##         - false positives
##         - false negatives
##     - ROC + AUC
##     - Accuracy
##     - Precision
##     - sensitivity
#=================================================================


get_prediction = function(fitted_probs, threshold){
  predictions = ifelse(fitted_probs > threshold, 1, 0)
  return(predictions)
}

evaluate_predictions = function(labels, predictions, verbose = FALSE){
  ct = table(factor(x = labels, levels = c(0, 1)),
             factor(x = predictions, levels = c(0,1)))

  if(any(dim(ct) != 2)) stop("Labels or Predictions contain more than 2 classes or
    both consist of only one class.")

  # Unpack values from table (for easier reference):
  TN = ct[1, 1]
  TP = ct[2, 2]
  FP = ct[1, 2]
  FN = ct[2, 1]

  # Calculate measures
  reports = list(
    sensitivity = TP / (TP + FN),
    specificity = TN / (FP + TN),
    precision = TP / (TP + FP),
    accuracy = (TP + TN) / sum(ct))

  if(verbose) print(data.frame(reports), digits = 3)
  return(reports)
}

evaluate_model = function(fitted_probs, labels){
  threshold_list = seq(from = 0, to = 1, by = 0.0001)
  pred_list = lapply(threshold_list,
                     get_prediction,
                     fitted_probs = fitted_probs)
  report_list = lapply(pred_list,
                       evaluate_predictions,
                       labels = labels)
  reports = unlist(report_list)
  # Calculate ROC:
  sensitivities = reports[names(reports) == "sensitivity"]
  specificities = reports[names(reports) == "specificity"]
  # Calculate AUC:
  get_auc = function(x, y){
    abs(sum(diff(x) * (head(y, -1) + tail(y, -1)))/2)}
  auc = get_auc(1-specificities, sensitivities)
  # Choose optimal metrics:
  opt_ind = which.max(sensitivities + specificities - 1)
  opt_sensitivity = sensitivities[opt_ind]
  opt_specificity = specificities[opt_ind]
  opt_threshold = seq(from = 0, to = 1, by = 0.0001)[opt_ind]
  opt_predictions = get_prediction(fitted_probs = fitted_probs,
                                   threshold = opt_threshold)
  opt_accuracy = (ct[1, 1] + ct[2, 2]) / sum(ct)
  opt_precision = precision = ct[2, 2] / (ct[2, 2] + ct[1, 2])
  ct = table(factor(x = labels, levels = c(0, 1)),
             factor(x = opt_predictions, levels = c(0,1)))
  plot(x = 1 - specificities, y = sensitivities,
       main = "ROC-Curve",
       xlab = "False Positive Rate (1 - specificity)",
       ylab = "True Positive Rate (sensitivity)",
       xlim = c(0, 1),
       ylim = c(0, 1),
       asp = TRUE,
       type = "s")
  abline(c(0, 0), c(1,1), col = "grey")
  return(list(sensitivity = opt_sensitivity,
              specificity = opt_specificity,
              accuracy = opt_accuracy,
              precision = opt_precision,
              threshold = opt_threshold,
              predictions = opt_predictions,
              auc = auc,
              contingency_table = ct))}
```

References

[1] Berk, J., DeMarzo, P. (2016), Corporate Finance, 4th edition, Harlow, Pearson Education

[2] Chen, S., Härdle, W.K., Moro, A.R (2011), Modeling default risk with support vector machines, Quantitative Finance, 11(1), 135-154

[3] Duda, R. O., Hart, P. E., Stork, D. G. (1973). Pattern Classification. New York: Wiley-Interscience.

[4] Fawcett, T. (2006), An introduction to ROC analysis, Pattern Recognition Letters 27, 861-874

[5] Härdle, W.K., Prastyo, D.D., Hafner, C. (2012), Support vector machines with evolutionary model selection for default prediction, SFB 649 Discussion Paper 2012-030

[6] Härdle, W. K., Simar, L., (2003). Applied Multivariate Statistical Analysis. Heidelberg: Springer-Verlag.

[7] Kabacoff, R. I. (2011). R in Action - Data Analysis and graphics with R. Shelter Island, NY 11964: Manning Piblucations Co..

[8] Maindonald, J. & Braun, J. W. (2010). Data Analysis and Graphics Using R - an Example-Based Approach. NY: Cambrigde University Press.

[9] Miao, H., Ramchander, S., Ryan, P., Wang, T. (2018), Default prediction models: The role of forward-looking

[10] Winkelmann, R., Boes, S. (2009): *"Analysis of Microdata"*, 2nd edition. measures of returns and volatility, Journal of Empirical Finance, 46, 146-162

[11] Zhang, J.L., Härdle, W.K. (2010). The Bayesian Additive Classification Tree Applied to Credit Risk Modelling. Computational Statistics and Data Analysis, 54: 1197-1205

[12] Zumel, N. & Munt, J. (2014). Practical Data Science with R. Shelter Island, NY 11964: Manning Publications Co..

[13] Breiman, L., Cutler, A., Liaw, A. & Wiener, M. (2015). Package 'randomForest', s.l.: s.n.

[14] Kuhn, M. et al. (2017). Package 'caret', s.l.: s.n.

[15] Milborrow, S. (2016). Plotting rpart trees with the rpart.plot package, s.l.: s.n.

[16] Modi, M. (2016). Different random forest packages in R, s.l.: s.n.

[17] Berend, H. (2017). Package 'geigen', s.l.: s.n.

[18] Wickham, H. (2017). Package 'dplyr', s.l.: s.n.

[19] Modi, M. (2016). linkedin. [Online] Available at: https://www.linkedin.com/pulse/different-random-forest-packages-r-madhur-modi/

[20] Rickert, J. (2013). Revolutions. [Online] Available at: http://blog.revolutionanalytics.com/2013/06/plotting-classification-and-regression-trees-with-plotrpart.html

[21] Robinson, N. (2018). The Disadvantages of Logistic Regression. [Online] Available at: https://classroom.synonym.com/disadvantages-logistic-regression-8574447.html

[22] Stephens, T. (2016). Trevor Stephens. [Online] Available at: http://trevorstephens.com/kaggle-titanic-tutorial/r-part-5-random-forests/

[23] Strobl, C., Boulesteix, A.-L., Zeileis, A. & Hothorn, T. (2007). BMC Bioinformatics. [Online] Available at: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-8-25