1 Binary Response Models

1.1 Theory

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

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

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

$$P(y=1|x) = G(x\prime\beta), \qquad \beta \in \mathbb{R}^d, \tag{2}$$

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

$$P(y=1|x) = G(x\prime\beta) = \Phi(x\prime\beta) = \int_{-\inf}^{x\prime\beta} \frac{1}{\sqrt{2\pi}} exp\left[-(\frac{t^2}{2})\right] dt. \tag{3}$$

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 $\Lambda(x\prime\beta)$:

$$P(y=1|x) = G(x\prime\beta) = \Lambda(x\prime\beta) = \frac{exp(x\prime\beta)}{1 + exp(x\prime\beta)} \tag{4}$$

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

$$\begin{aligned} L(\beta; y, x) &= \prod_{i=1}^{n} f(y_i|x_i) = \prod_{i=1}^{n} P(y_i=1|x_i)_i^y \left(1 - P(y_i=1|x_i)\right)^{1-y_i} \\ &= \prod_{i=1}^{n} G(x_i\prime\beta)_i^y \left(1 - G(x_i\prime\beta)\right)^{1-y_i} \end{aligned} \tag{5}$$

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

The log-Likelihood can thus be written as

$$l = \log L(\beta; yx) = \sum_{i=1}^{n} y_i \log G(x_i\prime\beta) + (1-y_i)\log(1 - G(x_i\prime\beta)) \tag{6}$$

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

$$\beta_M L = argmax(l) \tag{7}$$

and solve the first order conditions for a maximum.

$$\frac{\partial l}{\partial \beta} \stackrel{!}{=} 0 \tag{8}$$

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, one we will implement in the next section.

1.2 Implementation

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 very easily obtain predictions by invoking the `predict()`-function, which has been augmented with a method for the `brm`-class.

1.2.1 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 6 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 1 get_likehood()

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

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

```
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 6. It then computes the scalar product between the regressors of `Xy_mat`'s row, which plays the role of $x_i\prime\beta$ in 6. 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()`.

```
30  distr = switch (mode,
31                  "logit" = plogis,
32                  "probit" = pnorm
33  )
```

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.

1.2.2 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

$$x_{t+1} = x_t - \eta \cdot \nabla f(x_t), \qquad t \in \mathbb{N}, \eta \in \mathbb{R}^+ \tag{9}$$

where η is the learning rate. 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

$$\nabla f(x_t) \approx \frac{f(x_{t+1}) - f(x_t)}{\epsilon}, \qquad \epsilon > 0 \tag{10}$$

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 $[-100, 100]$.

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.

Algorithm 2 gradientDescentMinimizer()

procedure SET UP AUXILIARY VARIABLES
    learn_rates ← descending sequence from `learn` to 0
    a ← matrix of 1000 randomly initialized points
    f_a ← vector of function values for each element of a
    update:a ← argmin(f_a)
    gradient ← **compute gradient evaluated at a**
    $i \leftarrow 0$
procedure PERFORM GRADIENT DESCENT
    $l \leftarrow 0$
    while: $i \leq max\_iter$ and any element of gradient > 0 :
        update : $a = a - \texttt{learn\_rates[i]} \cdot gradient$
        calculate : *gradient* = **calculate gradient**
    if i = max_iter **then raise** warning
    return : a

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.

```
143  a = matrix(data = runif(1000 * n_pars,
144                          min = -10,
145                          max = 10),
146             ncol = n_pars)
147  f_a = apply(a, 1, obj)
148  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.

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 with which it is easy to compute the gradients as their element-wise difference, normalized by `epsilon_step`. The gradients are finally trimmed if necessary and returned.

```
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
```

```
    D_trimmed = ifelse(abs(D) <= 100, abs(D), 100) * sign(D)
    return(D_trimmed)}
```

Algorithm 3 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** $anyd \in D \notin [-100; 100]$ **then replace** d by $100 \cdot \text{sign(d)}$

```
  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)
```

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 infinite times. If the current iteration is

larger than `max_iter` the algorithm will break and the user will receive a warning (lines 175-176). If the user wishes 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).

1.2.3 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.

```
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` 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".

2 Linear Discriminant Analysis

2.1 Theory

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 within class variance.

$$max\ J_b(w) = w\prime S_b w, \qquad w \in \mathbb{R}^d, d \in \mathbb{N} \tag{11}$$

$$min\ J_w(w) = w\prime S_w w \tag{12}$$

This is done by maximizing the so called Raleigh coefficient

$$maxJ = \frac{J_b(w)}{J_w(w)} = \frac{w\prime S_b w}{w\prime S_w w}. \tag{13}$$

The matrices for between and within class variance are defined as

$$S_b = \sum_{c=1}^{C} (\mu_c - \mu)(\mu_c - \mu)\prime \tag{14}$$

$$S_w = \sum_{c=1}^{C} \sum_{i \in c} (x_i - \mu_c)(x_i - \mu_c)\prime \tag{15}$$

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

$$\arg\min_w -\frac{1}{2} w\prime S_b w \quad s.t. \quad w\prime S_w w = 1 \tag{16}$$

with the lagrangian being

$$\mathcal{L} = -\frac{1}{2} w\prime S_b w + \frac{1}{2}\lambda \left(w\prime S_w w - 1 \right). \tag{17}$$

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

$$S_b w = \lambda S_w w. \tag{18}$$

This is a generalized eigenvalue problem for which there exists a convenient `R`-solution in the form of the `geigen`-package.

2.2 Implementation

The result from 18, 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 18.

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

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

Algorithm 4 `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`

Unfortunately this is not exactly what we want for the scatter matrix defined in 15. 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.

```
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
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 18. This is done by the `geigen()` function from the `geigen` package.

```
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

$$intertia_i = \frac{\lambda_i^2}{\sum_{j=1}^{d} \lambda_j^2}. \tag{19}$$

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.

2.2.1 Plotting

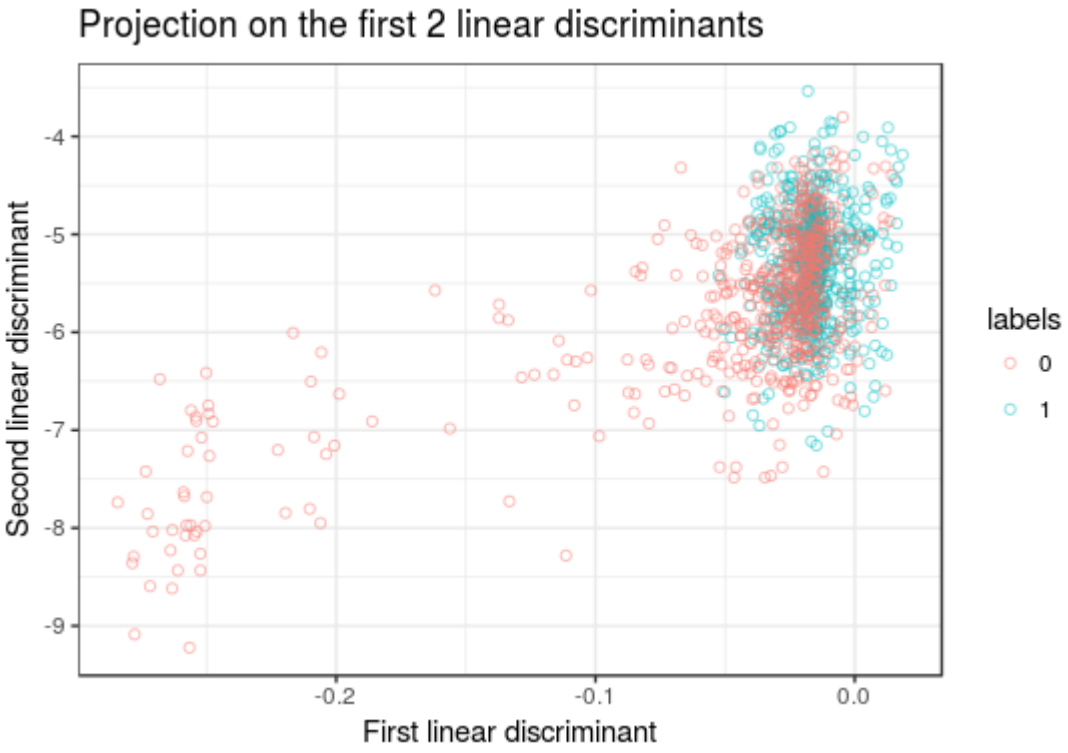
With a trained model of the `flda`-class it may be of interest to plot the rotations in order to get a visualisable idea of the separability of the two classes. To make this as easy as possible for the end-user we implemented a `plot` method for each instance of a `flda`-class. This means that a user 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`

```
61  plot.flda = function(model){
62    p = ggplot2::ggplot(data = model$X, ggplot2::aes_string(x = "lda1", y = "lda2",
        color = "labels")) +
63      ggplot2::geom_point(alpha = 0.5, shape=21) +
64      ggplot2::ggtitle("Projection on the first 2 linear discriminants") +
65      ggplot2::xlab("First linear discriminant") +
66      ggplot2::ylab("Second linear discriminant") +
67      ggplot2::theme_bw()
68    return(p)
69  }
```

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.

Figure 1: Using Linear Discriminant Analysis on the Creditreform Database



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.

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

2.2.2 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.

```
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)})
```

```
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

$$c = \frac{1}{2} v\prime (\mu_1 + \mu_2) \tag{20}$$

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.

3 Appendix

3.1 Unit Tests

3.1.1 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 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 a obviously visible separation line in figure 3.1.1 which discriminates the Setosa-species almost perfectly from the rest.

Should the `lda`-function work as intended we would expect this result only to become better, i.e. the points being perfectly separable.

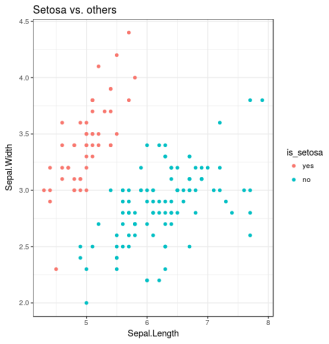


Figure 2: The Setosa-Species is already almost separable, even in the normal space.

3.1.2 Binary Response Models

3.1.3 Evaluation

References

[1] Winkelmann, R., Boes, S. (2009): *"Analysis of Microdata"*, 2nd edition.