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

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

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

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

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

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

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

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

[1]

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

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

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]

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.

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

```
15  l = function(x){
16    - sum(apply(Xy_mat, 1,
17                function(X){
18                  X[y_pos] * distr(t(X[-y_pos]) %*% x,
19                                   lower.tail = TRUE,
20                                   log.p = TRUE) +
21                    (1-X[y_pos]) * distr(t(X[-y_pos]) %*% x,
22                                         lower.tail = FALSE,
23                                         log.p = TRUE)}))}
24  # Return loglikelihood as a function of x (here 'x' stands for
      the weights)
25  return(l)
```

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 `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 $[-100, 100]$.

Algorithm 2 gradientDescentMinimizer()

procedure SET UP AUXILIARY VARIABLES
    learn_ rates ← descending sequence from `learn` to 0
    $a \leftarrow$ 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 : $a = a -$ `learn_rates[i]` $\cdot$ *gradient*
        calculate : *gradient* = **calculate gradient**
    if i = max_iter **then raise** warning
    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.

```
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
138    D_trimmed = ifelse(abs(D) <= 100, abs(D), 100) * sign(D)
139    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** $any d \in D \notin [-100; 100]$ **then replace** d by $100 \cdot \text{sign(d)}$

```
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)
178 }
```

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).

1.3 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".

1.4 Unit Tests

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

2.2 Implementation

2.3 Unit Tests

References

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