Machine Learning

Notes Part 2, HSG-MiQEF, Spring 2022

David Preinerstorfer

Last updated on 2022-03-02 12:52:45

Contents

Goals:

- Build a thorough understanding of classification algorithms.
- The focus in this notes is on **discriminant analysis** (LDA, QDA), **kernel based classification** (including the "naive Bayes classifier") and **regression based classification** (focusing on logistic regressions).

This lecture draws on on material from Chapters 3, 4 and 5 of Richter [2019] and Chapters 4 and 6 of Hastie et al. [2009].

1 Classification problems: revision and general structure

Recall from Lecture 1 that a supervised learning problem with the following structure is called a classification problem: the outcome Y is known to only take on a finite number of values $\{1, \dots, K\} \subseteq \mathbb{R}$.[1] In the classification setting, we use a loss function that is 0 if $y = f(\boldsymbol{x})$ and 1 otherwise. Formally, the loss function we use in such problems can be written as

$$L(y, s) = \begin{cases} 1 & \text{if } y \neq s \\ 0 & \text{else.} \end{cases}$$

We have already determined the optimal decision rule (i.e., the Bayes rule), which for the case of Y taking the labels $\{1, \dots, K\}$ is of the form

$$f^*(\boldsymbol{x}) \in \arg\min_{i=1,\dots,K} \mathbb{P}(Y \neq i | \boldsymbol{X} = \boldsymbol{x}) = \arg\max_{i=1,\dots,K} \mathbb{P}(Y = i | \boldsymbol{X} = \boldsymbol{x}).$$

Furthermore, note that the risk

$$\mathsf{R}(f) = \mathbb{E}(L(Y, f(\boldsymbol{X}))) = \mathbb{P}(Y \neq f(\boldsymbol{X})),$$

i.e., the risk equals the probability that we classify wrongly. The **training error** of a decision rule then is the proportion of misclassified data points in the training sample. Recall that if we have a testing sample, the estimated risk $\mathsf{empR}(\hat{f}_n)$ is just the proportion of misclassified data points in the testing sample.

An important observation that we will repeatedly use is the following:

> For every $i = 1, \dots, K$ we have (Bayes formula)
>
> $$\mathbb{P}\left(Y = i | \boldsymbol{X} = \boldsymbol{x}\right) = \frac{g_i(\boldsymbol{x})\pi_i}{\sum_{j=1}^K \pi_j g_j(\boldsymbol{x})}, \tag{1}$$
>
> where π_j denotes $\mathbb{P}(Y = j)$ and g_j denotes the density of $\boldsymbol{X}$ conditional on $Y = j$ so that $\sum_{j=1}^K \pi_j g_j(\boldsymbol{x})$ is the density of $\boldsymbol{X}$ by the law of total probability.

This tells us that we can formulate the probability that the observation Y is in class j given that $\boldsymbol{X} = \boldsymbol{x}$ in

[1] Note for later use that without any problem we can re-label these values. For example, if $K = 1$, instead of working with the values $\{1, 2\}$, it will sometimes be convenient to work with the values $\{-1, 1\}$ instead.

terms of the class-specific conditional distributions of the features and the class probabilities. In particular, we can write down the optimal decision rule in the following way (using the notation just introduced):

Optimal decision rule

$$f^*(\boldsymbol{x}) \in \arg\max_{i=1,\ldots,K} \frac{g_i(\boldsymbol{x})\pi_i}{\sum_{j=1}^K \pi_j g_j(\boldsymbol{x})} = \arg\max_{i=1,\ldots,K} g_i(\boldsymbol{x})\pi_i. \tag{2}$$

This is still a general result, because we have not put any assumptions on the densities of $\boldsymbol{X}$ given $Y = j$ yet.

2 Discriminant analysis

One of the oldest solutions to the classification problem is called **discriminant analysis** (DA). This method, which dates back to the 1930s and a statistician called R.A. Fisher, is based on a Gaussianity assumption that is put on the conditional distribution of the features $\boldsymbol{X}$ given $Y = i$. Given this assumption, we can then simplify the optimal decision rule given in Equation (2) and write it as functions of the parameters (mean vectors and covariance matrices) of the Gaussian distributions involved. A data-driven decision rule is then obtained by estimating these parameters from data. Recall from statistics that an estimator is nothing else than a function of the training sample to the respective parameter spaces. For Gaussian distributions, **maximum likelihood estimators** perform very well (which you may recall from your statistics course and a quantity called the Cramér-Rao bound).

2.1 Modeling assumption in the DA

The **modeling assumption** we impose in DA is that

$$\boldsymbol{X}|Y = j \sim \mathbb{N}_p(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \quad \text{for } j = 1, \ldots, K.$$

That is, the conditional distribution of the feature vector is assumed to be Gaussian. Note that this is an assumption, and whether it is a realistic one (at least approximately) or not depends on the context.[2]

You can find a synthetic data example for $K = 2$ (two classes) and $p = 2$ (two features) in Figure 1. Here, the parameters used were $\boldsymbol{\mu}_1 = (0, 0)'$, $\boldsymbol{\mu}_2 = (1, 1)'$ (the two mean vectors) and $\boldsymbol{\Sigma}_1 = \Sigma_2 = \boldsymbol{I}_2$ (i.e., the covariance matrices are the identities in both cases, which is particularly simple choice). Synthetic data has the advantage that we *know* the underlying distribution and can hence get an idea of what the data

[2] Also remember that an algorithm derived from such assumptions can be used on data regardless of whether the assumption holds or not. Its appropriateness can then be judged by comparing the algorithm's performance with the performance of competing methods by evaluating them on a testing sample.

corresponding to a certain parameter looks like. We can thus develop a feeling for what the parameters actually "mean" and how data generated from the distribution may look like (at least in case $p = 2$). We also show the code used for generating the data, so that you can play around with it by changing the parameters to see what happens (in the program the groups are coded as 0 and 1 instead of 1 and 2, in case you wonder).

```
set.seed(1)
y <- rbinom(250, 1, prob = 1/2)
X <- matrix(rnorm(250*2), nrow = 250, ncol = 2)
X[y==1,] <- X[y==1,] +1
plot(X, xlab = "X1", ylab = "X2")
points(X[y == 0,], col = "red")
points(X[y == 1,], col = "blue")
```

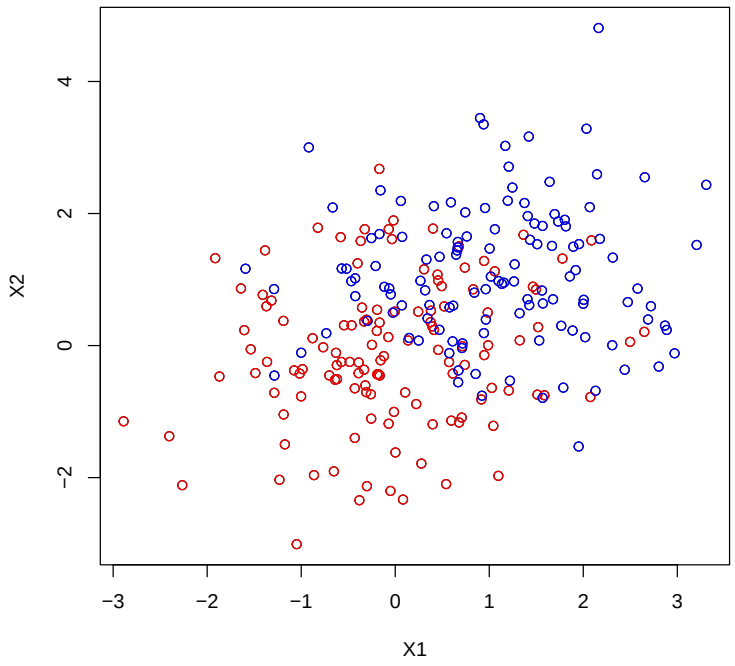

Figure 1: Data for a classification problem satisfying the assumptions imposed for discriminant analysis.

In Figure 1, the blue colored points correspond to group 2 and the red points correspond to group 1.

2.2 Optimal decision rules under the DA modeling assumption

In order to obtain the optimal decision rule, we just need to plug in the conditional densities into the rule we derived in (2). To this end, denote the density of a p-variate Gaussian with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ by

$$\phi_{\boldsymbol{\mu},\boldsymbol{\Sigma}} = (2\pi)^{-p/2}\det(\boldsymbol{\Sigma})^{-1/2}\exp\left[\frac{-1}{2}(\boldsymbol{x}-\boldsymbol{\mu})'\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right]. \tag{3}$$

From (2) we then obtain the optimal decision rule by setting $g_i = \phi_{\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i}$. That is we use the assumption that the conditional densities are Gaussian. We thus see that

$$f^*(\boldsymbol{x}) \in \arg\max_{i=1,\dots,K} \mathbb{P}(Y=i|\boldsymbol{X}=\boldsymbol{x}) = \arg\max_{i=1,\dots,K} \phi_{\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i}(\boldsymbol{x})\pi_i = \arg\max_{i=1,\dots,K} \log(\phi_{\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i}(\boldsymbol{x})\pi_i).$$

Here we take the logarithm, because this will result in a simpler expression for the decision rule. Note that the argument in the maximum remains unchanged if we apply a strictly monotonic transformation to the function that is to be optimized.

If we now use the expression from Equation (3), we see that

$$\log(\phi_{\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j}(\boldsymbol{x})\pi_j) = \frac{-p}{2}\log(2\pi) - \frac{1}{2}\log(\det(\boldsymbol{\Sigma})) - \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_j)'\boldsymbol{\Sigma}_j^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_j).$$

Because $\frac{-p}{2}\log(2\pi)$ does not depend on j, we can drop this term, and can thus conclude that:

Optimal decision rule for discriminant analysis under the Gaussianity assumption

$$f^*(\boldsymbol{x}) = \arg\max_{i=1,\dots,K}\left[-\frac{1}{2}\log(\det(\boldsymbol{\Sigma}_i)) - \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i)'\boldsymbol{\Sigma}_i^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_i) + \log(\pi_i)\right]$$

Note that in case we additionally impose the condition that all covariance matrices $\boldsymbol{\Sigma}_i$ be equal to $\boldsymbol{\Sigma}$, say, we can further drop some terms in the optimization problem defining the optimal decision rule that do not depend on i. Doing that results in the following optimization problem

Optimal decision rule for discriminant analysis under the Gaussianity assumption and assuming identical covariance matrices

$$f^*(\boldsymbol{x}) = \arg\max_{i=1,\dots,K}\left[\boldsymbol{x}'\boldsymbol{\Sigma}\boldsymbol{\mu}_i - \frac{1}{2}\boldsymbol{\mu}_i'\boldsymbol{\Sigma}_i^{-1}\boldsymbol{\mu}_i + \log(\pi_i)\right].$$

In our example that led to Figure 1, note that $\det(\boldsymbol{I}) = 1$ and $\boldsymbol{I}^{-1} = \boldsymbol{I}$, so that

$$f^*(\boldsymbol{x}) = \arg\max_{i=1,\dots,K}\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i)'(\boldsymbol{x}-\boldsymbol{\mu}_i) + \log(1/2)\right] = \arg\min_{i=1,\dots,K}(\boldsymbol{x}-\boldsymbol{\mu}_i)'(\boldsymbol{x}-\boldsymbol{\mu}_i) = \arg\min_{i=1,\dots,K}\|\boldsymbol{x}-\boldsymbol{\mu}_i\|^2.$$

This has a very simple interpretation: We assign $\boldsymbol{x}$ to class 1 if the distance from $\boldsymbol{x}$ to $\boldsymbol{\mu}_1$ is smaller (or equal to) the distance from $\boldsymbol{x}$ to $\boldsymbol{\mu}_2$. Let's take a closer look at that rule: the statement

$$(\boldsymbol{x} - \boldsymbol{\mu}_1)'(\boldsymbol{x} - \boldsymbol{\mu}_1) \geq (\boldsymbol{x} - \boldsymbol{\mu}_2)'(\boldsymbol{x} - \boldsymbol{\mu}_2)$$

rewrites

$$-2\boldsymbol{x}'\boldsymbol{\mu}_1 + \boldsymbol{\mu}_1'\boldsymbol{\mu}_1 \geq -2\boldsymbol{x}'\boldsymbol{\mu}_2 + \boldsymbol{\mu}_2'\boldsymbol{\mu}_2,$$

which for our example (recall that $\boldsymbol{\mu}_1 = (0, 0)'$, $\boldsymbol{\mu}_2 = (1, 1)'$) reduces to

$$0 \geq -2(x_1 + x_2) + 2 \quad \text{or equivalently} \quad x_1 + x_2 \geq 1,$$

which is shown as the region colored gray in Figure 2. We emphasize here that the "boundary functions" of the optimal rule, while being linear in that example, need not be linear in general (in our example, this is a consequence of the covariance matrices being *both* the identity matrix).

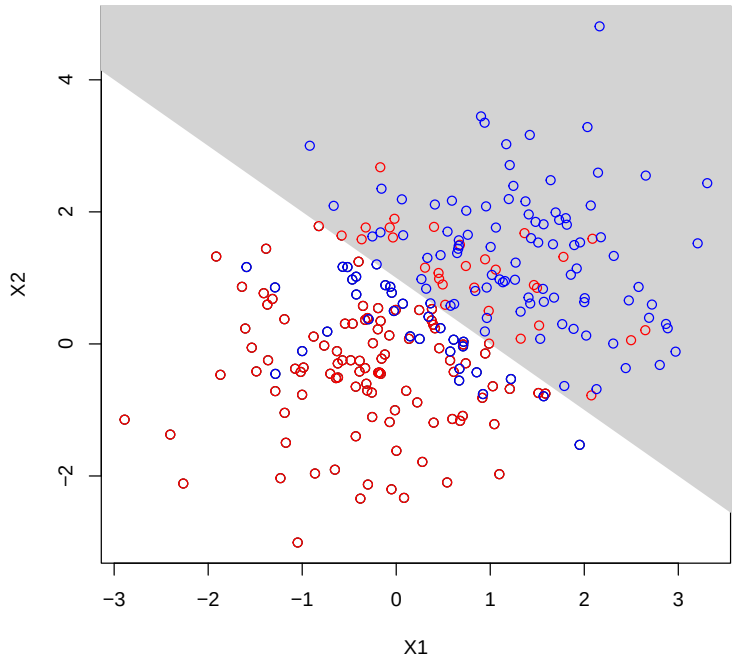

Figure 2: Optimal decision rule for a classification problem satisfying the assumptions imposed for discriminant analysis.

2.3 Classification rules: LDA and QDA

The optimal decision rule in the previous section depends on $\boldsymbol{\mu}_j$ and $\boldsymbol{\Sigma}_j$ for $j = 1, \ldots, k$ and also π_j. In practice, we obviously do not know these quantities; i.e., they are unknown parameters. The basic idea is now to estimate the parameters that show up in the optimal decision rule and replace the unknown quantities by the estimates to obtain a data-driven decision rule. More precisely, we use the following estimates (which, in fact, are the maximum likelihood estimates, but showing this is a bit tedious), where we denote the number of outcomes in the training sample that equal j by n_j for $j = 1, \ldots, K$:

- $\hat{\pi}_j = n_j/n$.
- $\hat{\boldsymbol{\mu}}_j = \frac{1}{n_j} \sum_{i:Y_i=j} \boldsymbol{X}_i$
- $\hat{\boldsymbol{\Sigma}}_j = \frac{1}{n_j} \sum_{i:Y_i=j} (\boldsymbol{X}_i - \hat{\boldsymbol{\mu}}_i)(\boldsymbol{X}_i - \hat{\boldsymbol{\mu}}_i)'$.

The "quadratic discriminant analysis" classifier is then defined as

Quadratic discriminant analysis classifier (QDA)

$$f^*_{qda}(\boldsymbol{x}) = \arg \max_{i=1,\ldots,K} \left[-\frac{1}{2} \log(\det(\hat{\boldsymbol{\Sigma}}_i)) - \frac{1}{2} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_i)' \hat{\boldsymbol{\Sigma}}_i^{-1} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_i) + \log(\hat{\pi}_i) \right]$$

The name "quadratic" already suggests that there is also a linear one. This classifier is obtained if one introduces the additional assumption that the covariance matrices $\boldsymbol{\Sigma}_j$ all coincide (but not the expectations $\boldsymbol{\mu}_j$, obviously). Then, one can use the full sample to estimate this common covariance matrix

- $\hat{\boldsymbol{\Sigma}}_j = \hat{\boldsymbol{\Sigma}} = \frac{1}{n} \sum_{i=1}^{K} n_k \hat{\Sigma}_i$.

With this choice one then obtains the classifier

Linear discriminant analysis classifier (LDA)

$$f^*_{lda}(\boldsymbol{x}) = \arg \max_{i=1,\ldots,K} \left[\boldsymbol{x}' \hat{\boldsymbol{\Sigma}}^{-1} \hat{\boldsymbol{\mu}}_i - \frac{1}{2} \hat{\boldsymbol{\mu}}_i' \hat{\boldsymbol{\Sigma}}^{-1} \hat{\boldsymbol{\mu}}_i + \log(\hat{\pi}_i) \right]$$

The LDA and QDA classifiers can be computed in `R` through the package **MASS**. The command for LDA is `lda(x, grouping, prior = proportions, tol = 1.0e-4, method, CV = FALSE, nu, ...)` and the one for QDA replaces `lda` by `qda`. The implementations allow us to apply the function to a subsample, do cross validation, etc. The following illustrates how `lda` (resulting in a classification region the boundary of which is linear) and `qda` (resulting in a classification region the boundary of which is "quadratic") can be used in the data example from above. We color the region the respective classifier assigns to group 2 in light

blue (over a grid). The classifier corresponding to LDA is shown in Figure 3, which is generated with the following commands.

```
library(MASS)
data.da <- data.frame("y" = y, "x1" = X[,1], "x2" = X[,2])
fit_lda <- lda(y ~ ., data = data.da)
pts <- seq(-6, 6, length = 750)
G_lda <- data.frame(expand.grid(pts, pts))
G_lda <- data.frame("x1" = G_lda[,1], "x2" = G_lda[,2])
pred_lda <- predict(fit_lda, newdata = G_lda)$class
plot(X, xlab = "X1", ylab = "X2")
points(G_lda[pred_lda == 1,], pch = 46, col = "lightblue", lwd = 1)
points(X[y == 0,], col = "red")
points(X[y == 1,], col = "blue")
```

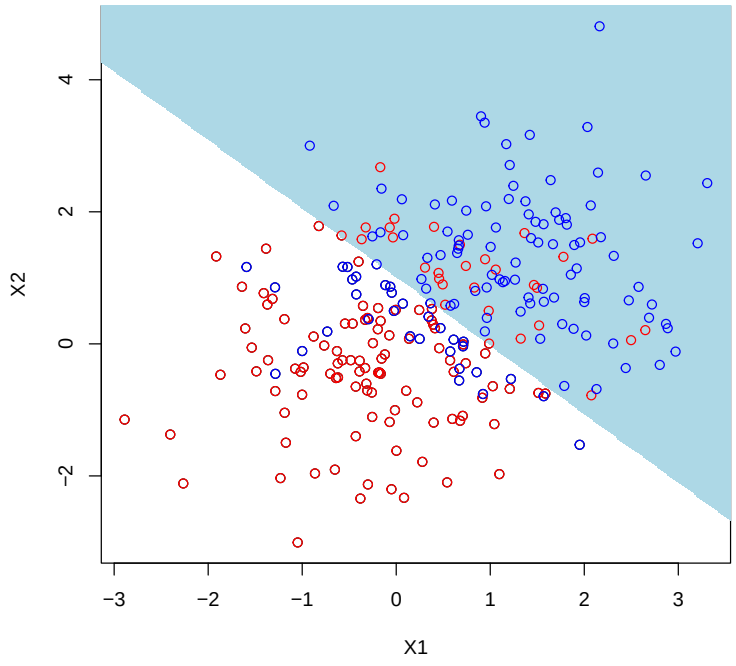

Figure 3: LDA classifier in our example.

The classifier corresponding to QDA can be seen in Figure 4, which is generated with the following lines of code.

```r
fit_qda <- qda(y ~ ., data = data.da)
pred_qda <- predict(fit_qda, newdata = G_lda)$class
plot(X, xlab = "X1", ylab = "X2")
points(G_lda[pred_qda == 1,], pch = 46, col = "lightblue", lwd = 1)
points(X[y == 0,], col = "red")
points(X[y == 1,], col = "blue")
```

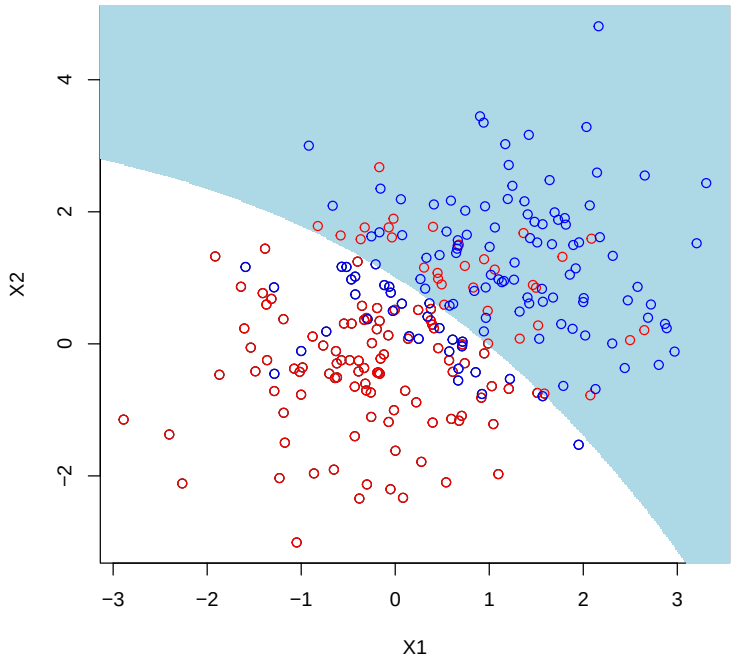

Figure 4: QDA classifier in our example.

Note that the blue regions are the classification regions that we learned from the training sample. They do not coincide 1:1 with the true optimal classification regions, simply because we make an error when estimating the parameters from the sample. But the region corresponding to `lda` is very close to the optimal one. Furthermore, note that the region obtained through `qda` is further off, but this is because `lda` uses the assumption (which is correct in the case considered) that the covariance matrices are equal. If they are not, then `qda` typically does a better job than `lda`.

We can also compute the training error, i.e., the proportion of misclassified data points by the two classifiers we just obtained. More informative even is what is called the confusion matrix, which reports predicted vs observed classes. We thus obtain not only the number of misclassified data, but the true positives (correctly

classified), true negatives (misclassified), false positives (misclassified), and false negatives (correctly classified). To obtain this in R, we just obtain the predictions from the lda and qda objects, and compare them with the true labels. We can proceed as follows:

```
#training error LDA
conf_lda <- table(list(predicted=predict(fit_lda)$class, observed=data.da$y))
conf_lda
```

```
##           observed
## predicted  0  1
##         0 96 33
##         1 30 91
```

```
sum(diag(conf_lda))/sum(conf_lda)
```

```
## [1] 0.748
```

```
#training error QDA
conf_qda <- table(list(predicted=predict(fit_qda)$class, observed=data.da$y))
conf_qda
```

```
##           observed
## predicted  0  1
##         0 94 32
##         1 32 92
```

```
sum(diag(conf_qda))/sum(conf_qda)
```

```
## [1] 0.744
```

Both methods perform similarly on the training sample. About 75% of the observations are correctly classified in the training sample. Recall that this is not a good estimate of the actual risk, because we used the same data for training and testing. To evaluate the models better we should work with a training and testing sample, which we skip for brevity here.

An application to classifying movements of a stock market index is discussed in detail here, an analysis that is taken from James et al. [2013].

Obviously, the data example we have taken a look at is very favorable for the method. This is simply because all the modeling assumptions are satisfied. Furthermore, it is quite obvious how the classification region

should look like, if we "look at the data" (visual inspection is a natural performance upper bound on any classification algorithm in dimension $p = 2$ and perhaps also $p = 3$, in the sense that whenever there is structure in the data, the human eye will typically find it easier to detect the signal than any algorithm; however, note that if $p > 3$ the situation becomes much more difficult and we then need to rely on algorithms).

To showcase a situation when classification methods break down (in the sense that they do not work that well) we can do the following: let's ry to do a similar classification analysis for predicting the (challenging) **bitcoin USD** course (you can download data via, e.g., https://finance.yahoo.com, the data we look at is taken from the past 365 days). The outcome is whether the bitcoin price in USD (over the last year) increased when compared to the day before (1) or did not increase (0), and the features are the changes in the prices in percent from the previous two days, respectively). You can see a scatterplot of the features in Figure 5. Here eyeballing suggests that the classes are not "well-separated", hence it will be very difficult to build a model that classifies well.

```
#load and prepare the data
btc <- read.table("btc.csv", sep = ",", dec = ".", header = TRUE)
btc[,3] <- log(btc[,3])
y_btc <- as.factor((btc[4:366,3]/btc[3:365,3]-1 > 0)*1)
x1_btc <- btc[3:365,3]/btc[2:364,3]-1
x2_btc <- btc[2:364,3]/btc[1:363,3]-1
btc_data <- data.frame("y" = y_btc, "x1" = x1_btc, "x2" = x2_btc)
X_btc <- cbind(x1_btc, x2_btc)
plot(X_btc, xlab = "X1", ylab = "X2")
points(X_btc[y_btc == 0,], col = "red")
points(X_btc[y_btc == 1,], col = "blue")
```

The reason to include this example here is not that we believe that one could come up with a good predictor for this dataset by simply using `lda` or `qda`. It is included to also take a look at an example where idealized conditions are not met. Pedagogically, this then allows us to see what the algorithms do in a very unfavorable situation (which is sometimes much more revealing than comparing them under idealized settings, in which essentially any algorithm will deliver the same result). The classification region obtained with `qda` is shown in Figure 6 and is obtained by applying the following code:

```
#fit a QDA
fit_btc <- qda(y ~ ., data = btc_data)
```

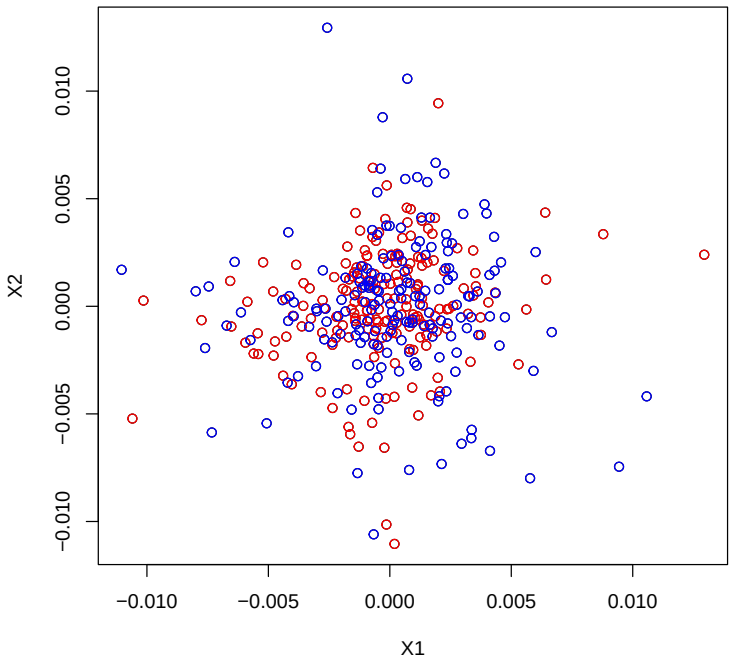

Figure 5: QDA classifier in our bitcoin example.

```r
#prepare grid for coloring
pts_btc <- seq(-0.02, 0.02, length = 1000)
G_btc <- data.frame(expand.grid(pts_btc, pts_btc))
G_btc <- data.frame("x1" = G_btc[,1], "x2" = G_btc[,2])
pred_qda_btc <- predict(fit_btc, newdata = G_btc)$class


#create plot
plot(X_btc, xlab = "X1", ylab = "X2")
points(G_btc[pred_qda_btc == 1,], pch = 46, col = "lightblue", lwd = 1)
points(X_btc[y_btc == 0,], col = "red")
points(X_btc[y_btc == 1,], col = "blue")
```

```r
#confusion matrix
conf_qda_btc <- table(list(predicted=predict(fit_btc )$class, observed=btc_data$y))
conf_qda_btc
```

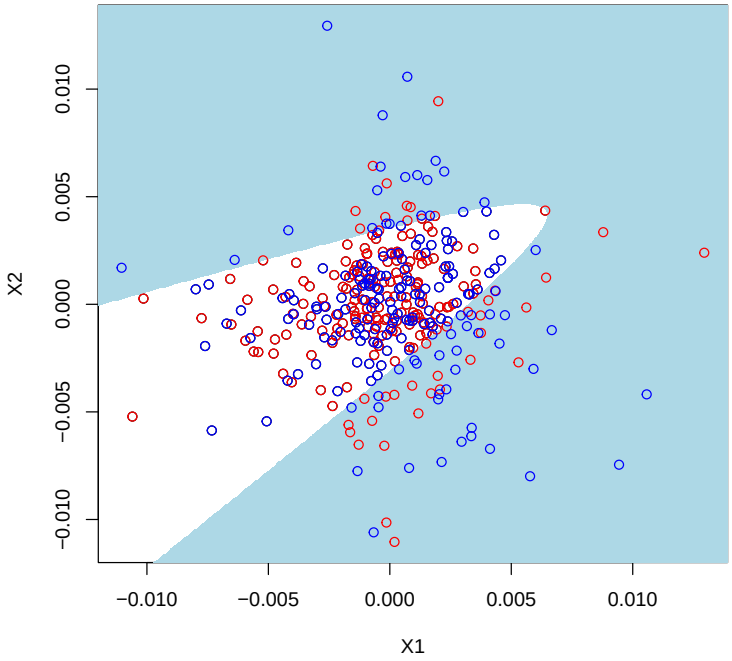

Figure 6: QDA classifier in our bitcoin example.

```
##           observed
## predicted   0   1
##         0 145 125
##         1  39  54
```

```
sum(diag(conf_qda_btc))/sum(conf_qda_btc)
```

```
## [1] 0.5482094
```

The classification we obtain, however, is quite reasonable.

3 Kernel density classification and the naive Bayes classifier

The approach we discuss here is a direct generalization of the approach underlying DA. There, we have assumed a *particular* **parametric** form of the conditional distribution of $\boldsymbol{X}$ given Y. More generally and flexibly, instead, we do not want to make an assumption concerning the conditional distributions. We want to keep them as unspecified as possible, i.e., **nonparametric**, the idea being that we can hence avoid mistakes that would arise if we misspecified this distribution.

Recall from Equation (2) that the optimal decision rule equals

$$f^*(\boldsymbol{x}) \in \arg\max_{i=1,\ldots,K} \frac{g_i(\boldsymbol{x})\pi_i}{\sum_{j=1}^K \pi_j g_j(\boldsymbol{x})}. \tag{4}$$

We now attempt to estimate this quantity directly. To this end, we need

1. Estimates $\hat{\pi}_j$ of π_j for $j = 1, \ldots, K$.
2. Estimates $\hat{g}_j$ of the conditional densities g_j for $j = 1, \ldots, K$.

Once we have those quantities we choose:

Kernel density classifier

$$\hat{f}_n(\boldsymbol{x}) \in \arg\max_{j=1,\ldots,K} \frac{\hat{g}_j(\boldsymbol{x})\hat{\pi}_j}{\sum_{i=1}^K \hat{\pi}_i \hat{g}_i(\boldsymbol{x})}. \tag{5}$$

Note that $\hat{\pi}_j$ can be estimated by taking the proportions of observations in the training sample for which $Y_i = j$ holds. This is simple and the estimates are good.

Estimating the conditional densities $g_j : \mathbb{R}^p \to \mathbb{R}$, however, is more complicated. It is more complicated, simply because these are **functions**. Therefore, we need to estimate K densities using the training sample. This can be done in various ways and is a major subject of the discipline of nonparametric statistics. We do not need much of the underlying theory however. The heuristic is explained in the following subsection.

3.1 Kernel density estimation: heuristics

Assume that $\boldsymbol{Z}_1, \ldots, \boldsymbol{Z}_N$ are i.i.d. random vectors with density g, say, which we want to estimate at the fixed point $\boldsymbol{z}$, say. Once we have an estimator at any point, we are done. The derivation is easier if we fix the point from the outset.

Fix a rectangle

$$\mathcal{N}(\boldsymbol{z}) := [z_1 - \lambda/2, z_1 + \lambda/2] \times \ldots \times [z_N - \lambda/2, z_N + \lambda/2]$$

of side-length λ, say, (and hence volume λ^N). By the definition of a density, it holds that

$$\mathbb{P}(\boldsymbol{Z}_i \in \mathcal{N}(\boldsymbol{z})) = \int_{\mathcal{N}(\boldsymbol{z})} g(\boldsymbol{z}) d\boldsymbol{z}.$$

If the density g does not vary too much for values close to $\boldsymbol{z}$, then, for small λ, it holds that

$$\mathbb{P}(\boldsymbol{Z}_i \in \mathcal{N}(\boldsymbol{z})) = \int_{\mathcal{N}(\boldsymbol{z})} g(\boldsymbol{z}) d\boldsymbol{z} \approx g(\boldsymbol{z}) \int_{\mathcal{N}} d\boldsymbol{z} = g(\boldsymbol{z})\lambda^N.$$

As a consequence, we *could* estimate $g(\boldsymbol{z})$ as

$$\tilde{g}(\boldsymbol{z}) = \frac{\#\{i : \boldsymbol{Z}_i \in \mathcal{N}(\boldsymbol{z}))\}}{N\lambda^N}.$$

That is: we count how many observations fall into the "bin" $\mathcal{N}(\boldsymbol{z})$ and re-scale that number by the volume of the bin (this certainly reminds you of a **histogram**, and that's precisely what it is!). The expectation of that estimator is clearly $\mathbb{P}(\boldsymbol{Z}_i \in \mathcal{N}(\boldsymbol{z}))/\lambda^N$, which is close to $g(\boldsymbol{z})$ if λ is small enough (note that if we choose λ too small, there will be very few or no observations in the bin, so we face a trade-off). This is not the final estimate, however. The problem is that if we vary $\boldsymbol{z}$ through all numbers, and take a look at how the estimates change as a function of $\boldsymbol{z}$, the resulting $\tilde{g}(\boldsymbol{z})$ would not be continuous (just as a histogram ...). But continuity, or some regularity, was assumed in the derivation of the estimate! Hence, it makes sense to "smooth" the values of the estimates we got.

To see how this can be achieved, let's rewrite the estimate once more: define the function $\boldsymbol{a} \mapsto K_\lambda(\boldsymbol{a}, \boldsymbol{z})$ that is $1/\lambda^N$ if the input to this function is in $\mathcal{N}(\boldsymbol{z}))$ and 0 else. We can then write

$$\tilde{g}(\boldsymbol{z}) = \frac{1}{N}\sum_{i=1}^{N} K_\lambda(\boldsymbol{Z}_i, \boldsymbol{z}).$$

Now, the problem is that the function K_λ is discontinuous. If $\boldsymbol{Z}_i$ is an element of $\mathcal{N}(\boldsymbol{z})$, then $K_\lambda(\boldsymbol{Z}_i, \boldsymbol{z}) = 1$, whereas it equals 0 otherwise. The trick is now to replace the function K_λ by something that is "smoother". That is, we use instead of that function, a smooth density $K(. - \boldsymbol{z})$, say.

This then results in the **kernel density estimator**

$$\hat{g}(\boldsymbol{z}) = \frac{1}{N}\sum_{i=1}^{N} K(\boldsymbol{Z}_i - \boldsymbol{z}).$$

The idea is that those $\boldsymbol{Z}_i$ that are close to $\boldsymbol{z}$ should get "more weight" than other observations. Popular kernels are the Gaussian kernel or the Epanechnikov kernel, but I won't go into detail. It is also intuitively clear that we should make the bin $\mathcal{N}(\boldsymbol{z}))$ smaller for larger values of observations in the training sample. This is achieved through a tuning parameter in the kernel (that is equal to its reciprocal "variability"). There is a multitude of automatic and non-automatic ways of choosing this parameter (which essentially all try to balance bias and variance). In `R` you can obtain kernel density estimates for a sample $z_1, \ldots, z_m$ through the function `density`.

Note that if we want to estimate a conditional density of $\boldsymbol{X}$ given that $Y = j$, we just need to use the above recipe with $\boldsymbol{Z}_1, \ldots \boldsymbol{Z}_m$ those vectors among $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ for which the corresponding value of Y_i equals j! Then, we can readily apply this to obtain the predictor in (5) above.

3.2 Naive Bayes classifier

Consider a situation where there are *many* predictors, i.e., where p is large (relative to sample size). Then, estimating the densities g_i is very difficult (remember that we are then attempting to estimate K functions in high dimensions with very few observations). In order to simplify the situation a bit, one then sometimes makes the simplifying assumption that the predictors $X_1, \ldots, X_p$, i.e., the coordinates of $\boldsymbol{X}$ are independent of each other. This implies that the conditional densities factorize into

$$g^i(\boldsymbol{x}) = g^{i1}(x_1) \times \ldots g^{iK}(x_K)$$

for every $i = 1, \ldots, K$. This simplifies the problem substantially, because we only need to estimate one-dimensional conditional densities (which we can do by the same logic as outlined in the previous section, as we considered the general vector-valued case, which in particular applies to the univariate situation).

The corresponding predictor is implemented in the `R` package **e1071** through the function `naiveBayes` which takes the form `naiveBayes(formula, data, laplace = 0, ..., subset, na.action = na.pass)` Here `x` are the predictors and `y` the output. To use a kernel density estimator, one sets `usekernel = TRUE`. The result for the DA data is obtained through

```
library(e1071)
data.NB <- data.da
data.NB$y <- as.factor(data.NB$y)
fit_NB <- naiveBayes(y ~ x1 + x2, usekernel = TRUE, data = data.NB)
```

and the resulting classifier is shown in Figure 7.

The naive Bayes classifier works quite well in this example. For the much more challenging example of bitcoin prices, the classifier is applied through

```
library(e1071)
data.NBbtc <- btc_data
data.NBbtc$y <- as.factor(data.NBbtc$y)
fit_NBbtc <- naiveBayes(y ~ x1 + x2, usekernel = TRUE, data = data.NBbtc)
```

and the classifier is shown in Figure 8. Interestingly, the obtained classifier would suggest that we should go long, unless the return two-days ago was essentially 0 and yesterday's return was negative. However, even the training error is quite poor!

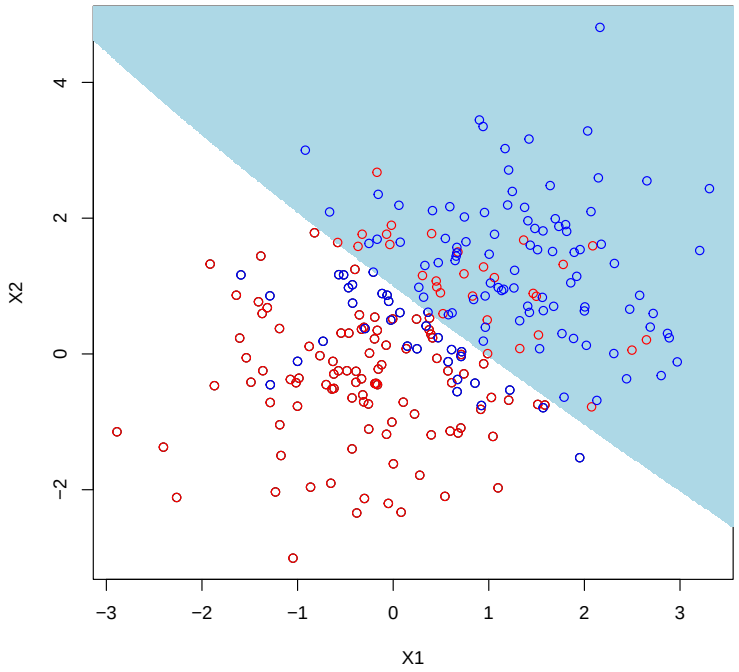

Figure 7: Naive Bayes classifier in the DA example.

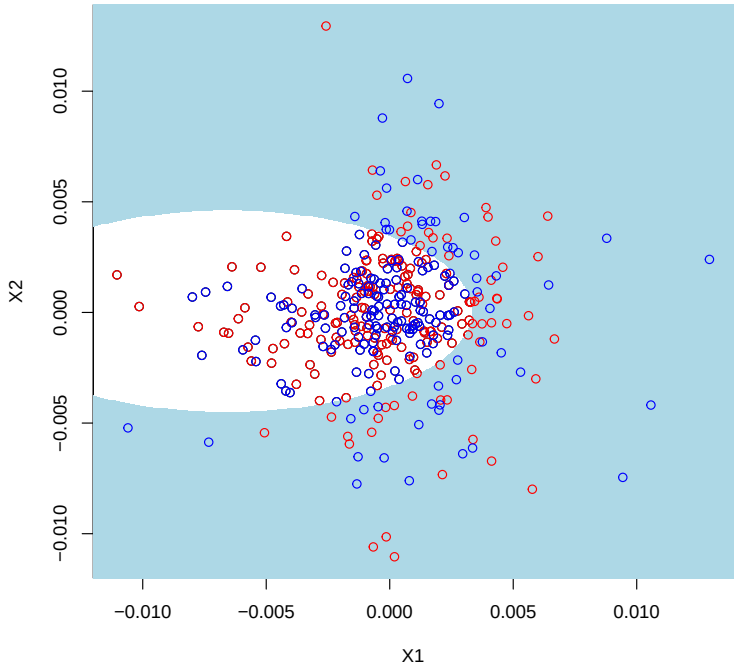

Figure 8: Naive Bayes classifier in the bitcoin price example.

4 Multiple logistic regression based classification

For the LDA and QDA we obtained the classifier through a plug-in approach based on estimating the quantities that determine the optimal decision rule. Underlying was a Gaussianity assumption on $\boldsymbol{X}$ given Y. We now consider a similar (multiple) logistic regression based classification (LRC) approach, that you are probably familiar with. Let's see how this fits into the picture.

4.1 Modeling assumption for LRC

The modeling assumption here is imposed on the conditional distribution $Y|\boldsymbol{X} = \boldsymbol{x}$ (and not on the conditional distribution of $\boldsymbol{X}$ given $Y = j$ as was done in the previous sections). Note that this is a distribution over the set $\{1, \ldots, K\}$ and that the distribution of $\boldsymbol{X}$ is not restricted; hence this method is flexible in that regard.

In logistic regression the distribution $Y|\boldsymbol{X} = \boldsymbol{x}$ is assumed to equal (note that the first predictor could be 1, i.e., the model "includes may include an intercept")

$$\mathbb{P}(Y = j|\boldsymbol{X} = \boldsymbol{x}) = \frac{\exp(\boldsymbol{x}'\boldsymbol{\beta}^j)}{1 + \sum_{i=1}^{K-1} \exp(\boldsymbol{x}'\boldsymbol{\beta}^i)} \quad \text{for } j = 1, \ldots, K-1 \tag{6}$$

and where the vectors $\boldsymbol{\beta}^i$ are unknown. Note that the probability $\mathbb{P}(Y = K|\boldsymbol{X} = \boldsymbol{x})$ is already determined from the above probabilities.

4.2 Optimal decision rule under the LRC modeling assumption

To derive the optimal decision rule in this setting, we have to determine

$$f^*(\boldsymbol{x}) \in \arg \max_{i=1,\ldots,K} \mathbb{P}(Y = i|\boldsymbol{X} = \boldsymbol{x}).$$

We can divide all probabilities by the probability from a reference category (since this then does not depend on i and take logarithms to write that equivalently as

$$f^*(\boldsymbol{x}) \in \arg \max_{i=1,\ldots,K} \log\left(\frac{\mathbb{P}(Y = i|\boldsymbol{X} = \boldsymbol{x})}{\mathbb{P}(Y = K|\boldsymbol{X} = \boldsymbol{x})}\right). \tag{7}$$

To simplify this further, we plug the expression for the conditional distribution (6) we assume into (7) to obtain

$$\log\left(\frac{\mathbb{P}(Y = i|\boldsymbol{X} = \boldsymbol{x})}{\mathbb{P}(Y = K|\boldsymbol{X} = \boldsymbol{x})}\right) = \boldsymbol{x}'\boldsymbol{\beta}^i.$$

Hence, the optimal decision rule under the modeling assumption we are currently working with equals

$$f^*(\boldsymbol{x}) \in \arg \max_{i=1,\ldots,K} \boldsymbol{x}'\boldsymbol{\beta}^i,$$

where we need to set $\boldsymbol{\beta}^K = \mathbf{0}$.

4.3 Classification rules for LRC through maximum likelihood

In order to learn the optimal decision rule from a training sample, we need to estimate the parameters $\boldsymbol{\beta}^i$ for $i = 1, \ldots, K-1$ that appear in the optimal decision rule. There are many ways to do that. The most important is probably through the method of maximum likelihood (which you are familiar with from the statistics courses you have taken). Note that the log-likelihood function is the expression

$$L(\boldsymbol{\beta}^1, \ldots, \boldsymbol{\beta}^{K-1}) = \sum_{i=1}^{n} \log\left(\mathbb{P}(Y = Y_i | \boldsymbol{X} = \boldsymbol{X}_i)\mathbb{P}_X(\boldsymbol{X}_i)\right) = \sum_{i=1}^{n} \log\left(\mathbb{P}(Y = Y_i | \boldsymbol{X} = \boldsymbol{X}_i)\right) + \sum_{i=1}^{n} \log\left(\mathbb{P}_X(\boldsymbol{X}_i)\right).$$

If we want to maximize this expression in $(\boldsymbol{\beta}^1, \ldots, \boldsymbol{\beta}^{K-1})$ we can drop the second sum, and we *could* simplify the expression. But, in any case, we cannot solve the resulting optimization problem in closed form and therefore have to rely on numerical optimization routines (such as the ones discussed previous week; cf. also Section 4.4.1 in Hastie et al. [2009]). Optimizing the log-likelihood function then results in estimates $(\hat{\boldsymbol{\beta}}^1, \ldots, \hat{\boldsymbol{\beta}}^{K-1})$ which, in turn, gives the decision rule

$$\hat{f}_n(\boldsymbol{x}) \in \arg\max_{i=1,\ldots,K} \boldsymbol{x}'\hat{\boldsymbol{\beta}}^i,$$

where we set $\hat{\boldsymbol{\beta}}^K = \mathbf{0}$. Note that if $K = 2$, that is if there are only 2 values the outcome can take on, the decision rule amounts to fitting a (univariate) logistic regression model to the data (with outcomes coded as 0 and 1, for example). Relabel the outcome as 0 instead of 2, i.e., the outcomes take values in 0 and 1. We then see that the above approach classifies an observation $\boldsymbol{x}$ as 1 if $\boldsymbol{x}'\hat{\boldsymbol{\beta}} \geq 0$, which is equivalent to the estimated probability that $\mathbb{P}(Y = 1|\boldsymbol{x}) \geq 1/2$.

For a dataset, we can fit a (multiple) logistic regression model through the `R` function `glm` via the command `glm(formula, family='binomial')`. Let's see how this works in the dataset we simulated above for the DA. Figure 9 contains the output to the following code:

```
fit_LR <- glm(y~ . , family = "binomial", data = data.da)
pred_lda <- (predict(fit_LR, newdata = G_lda) >=0)*1
plot(X, xlab = "X1", ylab = "X2")
points(G_lda[pred_lda == 1,], pch = 46, col = "lightblue", lwd = 1)
points(X[y == 0,], col = "red")
points(X[y == 1,], col = "blue")
```

Note that the resulting classification rule is very similar to the one obtained from LDA. Both decision rules

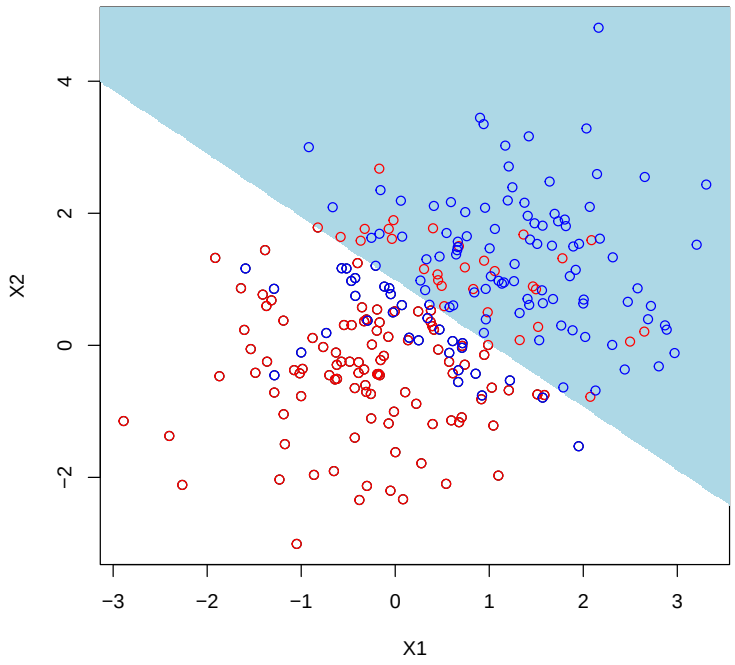

Figure 9: Logistic regression classifier in our DA example.

are "linear" (by definition) and are quite close to the optimal decision rule. We can repeat the exercise for the bitcoin data set, which results in Figure 10, which is generated via the following code:

```
fit_LR_btc <- glm(y ~ . , family = "binomial", data = btc_data)
pred_btcLR <- (predict(fit_LR_btc, newdata = G_btc) >=0)*1
plot(X_btc, xlab = "X1", ylab = "X2")
points(G_btc[pred_btcLR == 1,], pch = 46, col = "lightblue", lwd = 1)
points(X_btc[y_btc == 0,], col = "red")
points(X_btc[y_btc == 1,], col = "blue")
```

One can immediately work with a function $h(\boldsymbol{x})$ of the feature vector instead of the feature vector itself, which also applies to the methods discussed previously. One can also use penalized maximum likelihood estimators in case there are many features. In particular, when the number of features is larger than the size of the training sample! The potential problem, however, is that the number of variables in this optimization problem can be very large, much larger than n. In the section on support vector machines next week, we will encounter an approach that, while allowing for nonlinearities, never leads to optimization problems with more parameters than the size of the training sample!

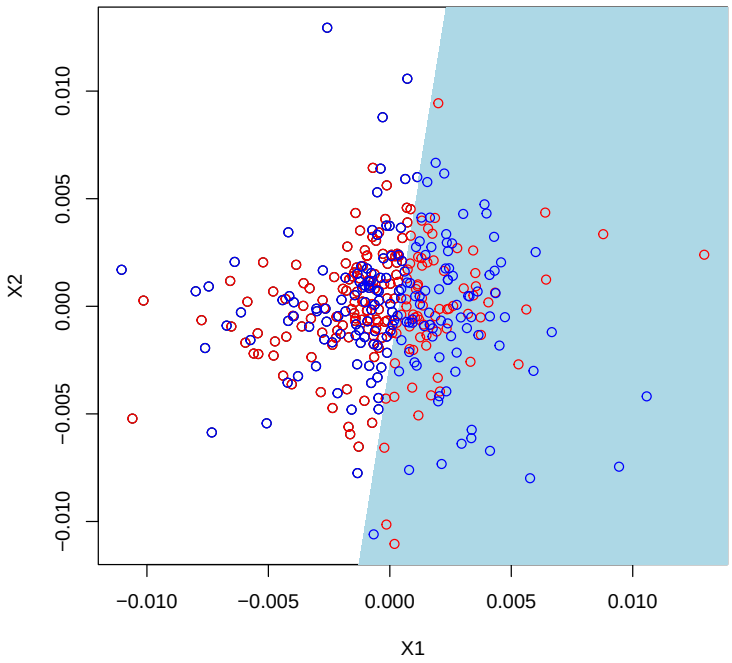

Figure 10: Logistic regression classifier in our bitcoin example.

Many variations of this regression-type approach are possible, but conceptually they are all very similar (and the penalized MLE estimators just mentioned can be obtained via the package **glmnet** as discussed in Lecture 1, but now using the binomial link function instead of the Gaussian one).

5 Exercise

By adding features of your choice (that are available to predict the next day's bitcoin price), you can try to improve the classification performance for the bitcoin price. Think about which features could be beneficial to include (also transformations of features are possible). Then, use two classification methods of your choice and try to build the best method for classification that you can come up with. To this end, divide the sample into a training and a testing sample, learn the models on the training sample and compare their performance on the testing part of the sample. Can you go beyond 60% accuracy in the testing sample? This exercise is worth a maximum of **5 points**.

References

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer: New York, 2009.

Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer: New York, 2013.

Stefan Richter. *Statistisches und maschinelles Lernen*. Springer: Berlin, 2019.