MTAT.03.227 Machine Learning (Spring 2014)
Exercise session VI: Linear classification

Konstantin Tretyakov

March 26, 2014

The aim of this exercise session is to get acquainted with the basics of linear classification methods. In particular, we are going to explore the following topics:

1. Algebra and geometry of *linear functionals* and *linear transformations*.
2. Fisher's linear discriminant.
3. The least squares approach to classification.
4. The perceptron algorithm.

For that we shall go through 15 exercises, each worth 1 point and presumably doable in under 20 minutes. For each exercise you typically need to write a short piece of code and perhaps a couple sentences of your opinion about what you did and saw. You can submit your whole solution as a single R file with comments, provided it is formatted to be sequentially readable and executable. As usual, the nominal point count you may aim for is 10, but I'm sure doing all 15pts won't hurt.

I provide you some base code to build your solutions upon (`linear_class.R`).

Linear Functionals

The theory of linear classification (as well as linear regression) revolves around the concept of a *linear*[1] *functional*[2]. A linear functional is simply a function $f : \mathbb{R}^m \to \mathbb{R}$ of the form:

$$f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = w_1x_1 + w_2x_2 + \cdots + w_mx_m.$$

A linear functional is uniquely defined by a *weight vector* $\mathbf{w}$ and in the following set of exercises we shall try to gain some intuition as to how it feels and looks like.

[1]In mathematics, a *linear* function is a function that satisfies $f(\alpha x + y) = \alpha f(x) + f(y)$.

[2]In mathematics, a *functional* is a general term which refers to a function which inputs a vector and outputs a scalar value.

The function `plot.classifier`, shown below (also present in the base code), takes as input a two-dimensional weight vector `w` and visualizes the corresponding linear functional $f_{\mathbf{w}}$ as a filled contour plot[3]

```
plot.classifier = function(w) {
  x1s = seq(-10, 10, 0.2)
  x2s = seq(-10, 10, 0.2)
  fvals = outer(x1s, x2s,
              Vectorize(function(x1, x2) { x1*w[1] + x2*w[2] }))
  image(x1s, x2s, fvals,
        col=terrain.colors(40), breaks=-20:20, asp=1)
  contour(x1s, x2s, fvals, levels=-40:40, add=T)
  arrows(0, 0, w[1], w[2], length=0.2, lwd=4)
}
```

Linear functional

Exercise 1 (1pt). Familiarize yourself with the code. Apply it on the weight vector `w = c(1, 0.5)` and study the resulting plot.

1. By just looking at the plot, guess the length of the weight vector $\mathbf{w}$.
2. Verify your guess by computing the actual length of $\mathbf{w}$ (show the code for computing the length).
3. By just looking at the plot, guess two arbitrary points $\mathbf{x}$ which would have $f_{\mathbf{w}}(\mathbf{x}) \approx 3.5$. Add them to the plot.

   Hint: Use `points(x[1], x[2])` for adding a point `x` to the plot.
4. Verify your guesses by computing actual $f_{\mathbf{w}}(\mathbf{x})$ for the two points.
5. Guess how the picture will look like if you increase the length of $\mathbf{w}$ two-fold. What would be the values of $f_{\mathbf{w}}(\mathbf{x})$ for your selected points? Verify your guesses. Do you see that the reasoning you used to guess the length of $\mathbf{w}$ in step 1 might have been wrong?

As a side-note, this is a nice place to introduce you to R's *generic function* capabilities. Try this:

```
class(w) = "classifier"
plot(w)
```

Do you see how R automatically selects the `plot.<xxx>` function based on the `class` attribute of an object? If interested, read more using `help(class)`.

[3]Note, we do not use R's built-in function `filled.contour` because it is bad (it messes up the coordinate system of the plot).

Solution. The length of a vector **w**, also known as the *norm* of **w** is equal to:

$$\|\mathbf{w}\| = \sqrt{w_1^2 + w_2^2} = \sqrt{\mathbf{w}^T\mathbf{w}}.$$

The corresponding code in R is then:

```
> norm = function(v) sqrt(v %*% v)
> norm(c(1, 0.5))
         [,1]
[1,] 1.118034
```

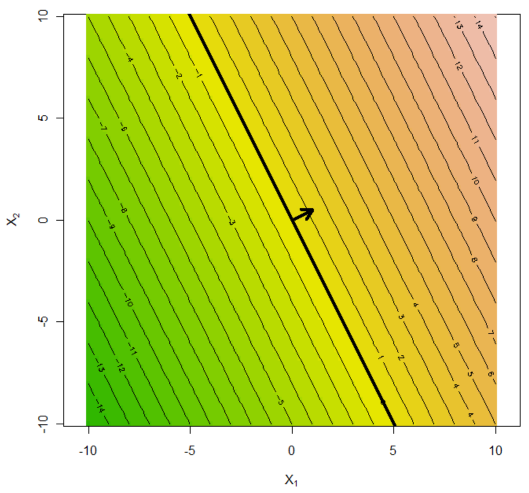
As the norm of **w** is increased, the function $f_\mathbf{w}$ becomes steeper (more level lines seen on the plot).

Note that if you try to measure the "length" of **w** in Step 1 by simply using the level lines as a ruler, you won't in general obtain the correct answer. Indeed, the value of $f_\mathbf{w}$ at point **w** is equal to $f_\mathbf{w}(\mathbf{w}) = \mathbf{w}^T\mathbf{w} = \|\mathbf{w}\|^2$, hence by counting level lines to the tip of **w** you will actually find the *square* of the length of **w**.

In Step 1, the choice of the vector **w** is such that the difference between $\|\mathbf{w}\|$ and $\|\mathbf{w}\|^2$ is small enough to be unnoticed by bare sight: it was meant to be misleading.

Exercise 2 (1pt). Modify the function `plot.classifier` to also draw a fat line, corresponding to all points where $f_\mathbf{w}(\mathbf{x}) = 0$. The result should look as follows:

Separating line



Hint: `abline(?, ?, lwd=4)`.

Solution. The function `abline(a, b)` will add to the plot the line

$$x_2 = a + bx_1$$

The line that we need is $f_{\mathbf{w}}(\mathbf{x}) = 0$, i.e.

$$w_1x_1 + w_2x_2 = 0,$$

which, after rearranging, becomes

$$x_2 = -\frac{w_1}{w_2}x_1,$$

hence for the purposes of `abline` the parameters are $a = 0$, $b = -\frac{w_1}{w_2}$. The necessary code line to be appended to the end of the `plot.classifier` function is just:

```
abline(0, -w[1]/w[2], lwd=4)
```

In the following exercise we shall extend this to also incorporate the bias term, i.e. the desired line will be $f_{(\mathbf{w},w_0)}(\mathbf{x}) = 0$, i.e.

$$w_0 + w_1x_1 + w_2x_2 = 0.$$

Rearranging this we get:

$$x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2}x_1,$$

and thus $a = -w_0/w_2$, $b = -w_1/w_2$.

Finally, the case $w_2 = 0$ should be handled separately. This corresponds to a vertical line at $x_1 = -w_0/w_1$. The full code is given in the solution to the next exercise.

Exercise 3 (1pt). Sometimes it is more convenient to regard a linear classifier as a function of the form[4]: *Bias term*

$$f_{(\mathbf{w},w_0)}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0.$$

Modify the function `plot.classifier` so that it would accept two parameters: $\mathbf{w}$ and w_0 (i.e. `plot.classifier = function(w, w0=0)`) and visualize the output of the corresponding function $f_{(\mathbf{w},w_0)}$. In particular, it should show the appropriate separating line using `abline`. You may keep the `arrows` invocation intact. Play with the visualization to get a feel at how w_0 affects the output.

1. Let `w = c(3, 4)`. Find `w0` such that the separating line passes through point `p = c(2.5, 0)`. What is the general formula?

   Hint: $f_{(\mathbf{w},w_0)}(\mathbf{p}) = 0$

[4]Formally, such a function is not a *linear functional* any more, this is an *affine functional*.

2. Find `w0` such that the separating line is at distance 2.5 from zero. What is the general formula?

   Hint: $\mathbf{p} = \frac{2.5\mathbf{w}}{\|\mathbf{w}\|}$ is at distance 2.5 from zero.

3. By how much exactly does the separating line shift from 0 for a given value of w_0? What is the general formula?

   Hint: The ordering of these three questions is intentional.

Solution. The modified `plot.classifier` function is presented below:

```
plot.classifier = function(w, w0=0) {
  x1s = seq(-10, 10, 0.2)
  x2s = seq(-10, 10, 0.2)
  fvals = outer(x1s, x2s,
        Vectorize(function(x1, x2) { x1*w[1] + x2*w[2] + w0 }))
  image(x1s, x2s, fvals,
        col=terrain.colors(40), breaks=-20:20, asp=1,
        xlab=expression(X[1]), ylab=expression(X[2]))
  contour(x1s, x2s, fvals, levels=-40:40, add=T)
  arrows(0, 0, w[1], w[2], length=0.2, lwd=4)

  if (w[2] != 0)
    abline(-w0/w[2], -w[1]/w[2], lwd=4)
  else if (w[1] != 0)
    abline(v = -w0/w[1], lwd=4)
}
```

1. In order to find a bias term w_0 such that the separating line passes through $\mathbf{p}$, we just solve $f_{(\mathbf{w},w_0)}(\mathbf{p}) = 0$:

$$w_0 + \mathbf{w}^T\mathbf{p} = 0$$
$$w_0 = -\mathbf{w}^T\mathbf{p}$$

   Let's test it:

```
w = c(3, 4)
p = c(2.5, 0)
w0 = -w %*% p
plot.classifier(w, w0)
points(p[1], p[2], cex=2, bg='red', pch=21)
```

2. The next task, finding a bias term such that the separating line is at distance 2.5 from zero, is somewhat more tricky. By looking hard at the plots you should note that the point on the separating line which is closest to the origin must be of the form $\mathbf{p} = c\mathbf{w}$. Consequently, if we shift the line so that it passes through $\mathbf{p} = (2.5/\|\mathbf{w}\|)\mathbf{w}$, the line will be at distance

2.5 as needed. Substituting this $\mathbf{p}$ into the formula $w_0 = -\mathbf{w}^T\mathbf{p}$ from the previous step we get

$$w_0 = -2.5\|\mathbf{w}\|.$$

3. From the previous equation we see that for a given w_0 the separating line shifts by distance $-w_0/\|\mathbf{w}\|$ from the origin. This observation suggests that using a *normalized* weight vector (i.e. $\mathbf{w}$ such that $\|\mathbf{w}\| = 1$) is sometimes convenient, because in this case $f_{(\mathbf{w},w_0)}(\mathbf{x})$ is exactly equal to the (signed) distance of the point $\mathbf{x}$ to the separating line.

   Let us now demonstrate the last two results more formally (and use it as a chance to illustrate constrained optimization). Consider a line

$$\mathbf{w}^T\mathbf{x} + w_0 = 0$$

   The point $\mathbf{p}$ on this line, closest to the origin, can be found by solving the following constrained optimization task:

$$\begin{aligned} \mathbf{p} = \ &\operatorname*{argmin}_{\mathbf{x}} \mathbf{x}^T\mathbf{x}, \\ &\text{s.t.} \quad \mathbf{w}^T\mathbf{x} + w_0 = 0. \end{aligned}$$

   By applying the method of Lagrange multipliers we have to define the Lagrangian

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda g_1(\mathbf{x}) = \mathbf{x}^T\mathbf{x} + \lambda(\mathbf{w}^T\mathbf{x} + w_0),$$

   and solve for

$$\nabla L(\mathbf{x}, \lambda) = 0.$$

   This condition translates to a system of equations:

$$\begin{cases} 2\mathbf{x} + \lambda\mathbf{w} & = \mathbf{0}, \\ \mathbf{w}^T\mathbf{x} + w_0 & = 0. \end{cases}$$

   The first equation resolves to what we noted in Step 2:

$$\mathbf{x} = -\frac{\lambda}{2}\mathbf{w} = c\mathbf{w} \text{ (for some } c) \ .$$

   and the second simply ensures the point must lie on the line. Substituting $c\mathbf{w}$ into the second equation we get:

$$\begin{aligned} c\mathbf{w}^T\mathbf{w} + w_0 = 0, \\ c = \frac{-w_0}{\|\mathbf{w}\|^2}, \end{aligned}$$

   and thus the solution point is:

$$\mathbf{x}_{\text{solution}} = c\mathbf{w} = \frac{-w_0}{\|\mathbf{w}\|^2}\mathbf{w}$$

The distance of this point to the origin (which is also the distance of the line to the origin is):

$$\|\mathbf{x}_{\text{solution}}\| = \|c\mathbf{w}\| = |c| \cdot \|\mathbf{w}\| = \frac{|w_0|}{\|\mathbf{w}\|^2} \cdot \|\mathbf{w}\| = \frac{|w_0|}{\|\mathbf{w}\|}.$$

We confirm here that if we choose $\mathbf{w}$ such that its norm is 1, the distance of the line to the origin is exactly $|w_0|$. The sign of w_0 indicates the direction of the shift. Positive w_0 means shifting *opposite* the direction of $\mathbf{w}$.

Linear Transformations

Whence a *linear functional* maps a vector to a number, a *linear transformation* maps a vector to a vector. A linear transformation of m-dimensional vectors is always an $m \times m$ matrix.

Exercise 4 (1pt). Generate and visualize a dataset of normally-distributed points:

Linear transformations

```
X = matrix(rnorm(100),ncol=2)
plot(X[,1],X[,2], asp=1)
```

1. Let $\mathbf{M}$ be a tranformation matrix. The application of this matrix to a point (i.e. column-vector) $\mathbf{x}$ can be computed using the expression $\mathbf{Mx}$. What is the correct expression to apply the transformation $\mathbf{M}$ to all rows of $\mathbf{X}$?

2. Define in R the following matrix:

$$\mathbf{R} = \begin{pmatrix} \cos(0.2) & -\sin(0.2) \\ \sin(0.2) & \cos(0.2) \end{pmatrix}.$$

   Apply it to all rows of dataset `X` to obtain a transformed dataset `Xt`. Add the transformed point to your plot as follows:

```
points(Xt[,1], Xt[,2], col='red')
segments(X[,1], X[,2], Xt[,1], Xt[,2], col='red')
```

3. Repeat the same for the following matrix:

$$\mathbf{S} = \begin{pmatrix} 3 & 0 \\ 0 & 0.5 \end{pmatrix}.$$

4. Finally, let $\mathbf{M} = \mathbf{RS}$. Try to guess, what $\mathbf{M}$ does to the points. Then verify your guess.

Solution.

1. The data matrix $\mathbf{X}$ has transposed points $\mathbf{x}^T$ as its rows. In order to apply matrix $\mathbf{M}$ we have to first transpose $\mathbf{X}$ (to get points into column-vectors), then apply $\mathbf{M}$ via usual left-multiplication, and finally transpose the result back. The total operation is then
$$(\mathbf{M}\mathbf{X}^T)^T.$$
By using the trasposition property $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$ we can simplify the expression to
$$\mathbf{X}\mathbf{M}^T.$$

2. The code for applying the matrix is:
```
R = matrix(c(cos(0.2),sin(0.2),-sin(0.2),cos(0.2)), nrow=2)
Xt = X %*% t(R)
```
The resulting operation is a rotation by 0.2 radians.

3. You should note that the operation $\mathbf{S}$ scales the points by factor 3 along the x_1 axis and shrinks twice along the x_2 axis.

4. Matrix multiplication is applied right-to-left, i.e.:
$$\mathbf{M}\mathbf{x} = (\mathbf{RS})\mathbf{x} = \mathbf{R}(\mathbf{S}\mathbf{x}).$$
Thus, matrix $\mathbf{M}$ first scales the points using $\mathbf{S}$, and then rotates using $\mathbf{R}$. Note that $\mathbf{RS} \neq \mathbf{SR}$!

Covariance normalization

Exercise 5-6 (2pt). The normally-distributed data X that we generated in the previous exercise adheres to the following probability law:
$$\Pr[\mathbf{x}] = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\mathbf{x}^T\mathbf{x}\right)$$
Now that we transform the points $\mathbf{x} \to \mathbf{M}\mathbf{x}$, it holds that:
$$\Pr[\mathbf{M}\mathbf{x}] \propto \exp\left(-\frac{1}{2}\mathbf{x}^T\mathbf{x}\right).$$
Now denote $\mathbf{x}' = \mathbf{M}\mathbf{x}$ and substitute into the above equation to obtain something of the form:
$$\Pr[\mathbf{x}'] \propto \exp\left(-\frac{1}{2}\mathbf{x}'^T\mathbf{\Sigma}^{-1}\mathbf{x}'\right).$$

1. Express $\mathbf{\Sigma}$ in terms of $\mathbf{M}$. Compute it in R from the $\mathbf{M}$ matrix.
2. It turns out that the *covariance* of the data is an estimate for $\mathbf{\Sigma}$. Verify that cov(Xt) is indeed close to what you just computed from $\mathbf{M}$.

3. The above should convince you that the matrix $\mathbf{\Sigma}^{-1/2}$ (either the true one or estimated from data using `cov`) will "untransform" the points back to a uniform cloud. Let us test it. To compute the square root of a matrix[5] please use the function below (available in the base code):

```
matrix_sqrt = function(M) {
  e = eigen(M)
  e$vectors %*% sqrt(diag(e$values)) %*% t(e$vectors)
}
```

Verify that `Xt %*% t(solve(matrix_sqrt(cov(Xt))))` is indeed a uniform point cloud. What we have essentially performed here is also known as *principal components analysis (PCA)*.

Solution.

1. If $\mathbf{x}' = \mathbf{M}\mathbf{x}$ then $\mathbf{x} = \mathbf{M}^{-1}\mathbf{x}'$. Substituting this into

$$\Pr[\mathbf{M}\mathbf{x}] \propto \exp\left(-\frac{1}{2}\mathbf{x}^T\mathbf{x}\right).$$

we get:

$$\Pr[\mathbf{x}'] \propto \exp\left(-\frac{1}{2}(\mathbf{M}^{-1}\mathbf{x}')^T(\mathbf{M}\mathbf{x}')\right)$$
$$\Pr[\mathbf{x}'] \propto \exp\left(-\frac{1}{2}\mathbf{x}'^T(\mathbf{M}^{-1})^T\mathbf{M}^{-1}\mathbf{x}'\right)$$
$$\Pr[\mathbf{x}'] \propto \exp\left(-\frac{1}{2}\mathbf{x}'^T(\mathbf{M}^T)^{-1}\mathbf{M}^{-1}\mathbf{x}'\right)$$
$$\Pr[\mathbf{x}'] \propto \exp\left(-\frac{1}{2}\mathbf{x}'^T(\mathbf{M}\mathbf{M}^T)^{-1}\mathbf{x}'\right)$$

and thus $\mathbf{\Sigma} = \mathbf{M}\mathbf{M}^T$.

2. Verify that the matrices `cov(Xt)` and `M %*% t(M)` are indeed similar.

```
> cov(Xt)
         [,1]      [,2]
[1,] 9.073840 1.8759618
[2,] 1.875962 0.6706222
> M %*% t(M)
         [,1]      [,2]
[1,] 8.654642 1.7037052
[2,] 1.703705 0.5953582
```

[5]Note that a matrix square root is not uniquely defined, hence you should not expect that $\mathbf{\Sigma}^{1/2} = \mathbf{M}$.

Fisher's Linear Discriminant

In the following we shall work with the classic sample *MT Cars* dataset, which is readily available in R as `mtcars`. Use `help(mtcars)` to read more about it. We shall use the `qsec` and `mpg` features as our x_1 and x_2 and the `am` feature as the class label. We drop instances 5, 25, and 32 and normalize the features to zero mean. This is all performed using the following `load.data` function (present in the base code).

```
load.data = function() {
  data = mtcars[-c(5,25,32),]
  x1 = (data$qsec - mean(data$qsec)) # Subtract means
  x2 = (data$mpg - mean(data$mpg))
  y = 2*data$am - 1                  # {0, 1} --> {-1, 1}

  data = list(cbind(x1, x2), y)
  names(data) = c("X", "y")
  class(data) = "data"
  data
}
```

You are also supplied with a helpful `plot.data` method:

```
plot.data = function(data, add=F, cex=3) {
  lbl = (data$y+1)/2  # {-1..1} -> {0..1}
  if (add)
    points(data$X[,1],data$X[,2], bg=lbl, pch=21+lbl, cex=cex)
  else
    plot(data$X[,1], data$X[,2], bg=lbl, pch=21+lbl, cex=cex, asp=1)
  text(data$X[,1], data$X[,2], col=(1-lbl), cex=0.2*cex)
}
```

Finally, you will need the following function for highlighting points:

```
mark.point = function(x, y=1, bg='red', cex=3) {
  points(x[1], x[2], bg=bg, cex=cex, pch=21+(y+1)/2)
}
```

Exercise 7-9 (3pt). Familiarize yourself with the code. Make sure you can load and plot the data.

Fisher's discriminant

1. Compute the means of the positive and negative examples. Mark them on the plot using `mark.point`.

   Hint: Use `colMeans`.

2. Let $\mathbf{w} = \mathbf{m}_1 - \mathbf{m}_0$, where $\mathbf{m}_i$ are the class means. Plot the classifier defined by $\mathbf{w}$ (using `plot.classifier`). Add the data points to the plot (use `plot(data, add=T)`). Count (visually) how many points are misclassified.

3. Obviously, the problem is that the data is highly skewed. Compute the covariance matrix of the data `cov(data$X)` and use it to perform covariance normalization as in Exercise 5-6. Visualize the transformed data.

4. Now compute $\mathbf{w} = \mathbf{m}_1 - \mathbf{m}_0$ on the transformed data as before. Visualise and count the number of misclassified examples.

   Hint: To "zoom in" the transformed data on the plot, simply multiply all data by a constant (e.g. `data$X = 3*data$X`).

5. We just used `cov(data$X)` as estimate of $\mathbf{\Sigma}$. The original Fisher's discriminant algorithm suggests to estimate $\mathbf{\Sigma}$ as an average of two class-conditional covariances:

   ```
   sigma1 = cov(data$X[data$y==1,])
   sigma2 = cov(data$X[data$y==-1,])
   sigma = (sigma1 + sigma2)/2
   ```

   Use this estimate to perform covariance normalization, plot and see whether the new classifier is different.

6. Instead of transforming the data, we can instead transform the weight vector. It turns out that the proper way to compute $\mathbf{w}$ without having to transform the data is simply

   $$\mathbf{w} = \mathbf{\Sigma}^{-1}(\mathbf{m}_1 - \mathbf{m}_0).$$

   Compute this vector (on untransformed data and with $\mathbf{\Sigma}$ computed as in step 5). Visualize the classifier and the untransformed data.

7. To complete the implementation of the Fisher's discriminant, we have to choose the bias term w_0. The traditional choice is to have the point $\mathbf{p} = 0.5(\mathbf{m}_1 + \mathbf{m}_0)$ lie on the separating line. Find this w_0. Plot the final result. Mark the location of $\mathbf{p}$ using `mark.point` on the plot.

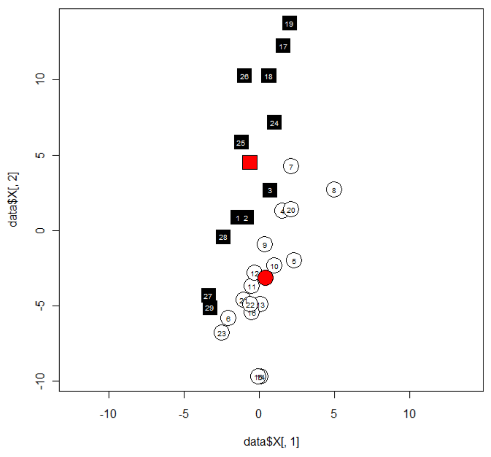
8. Congratulations, you have implemented Fisher's discriminant! Now compare your implementation to a library function:

   ```
   library(MASS)
   lda(data$X, data$y)$scaling
   ```

Solution.

1. The required plot can be done as follows:

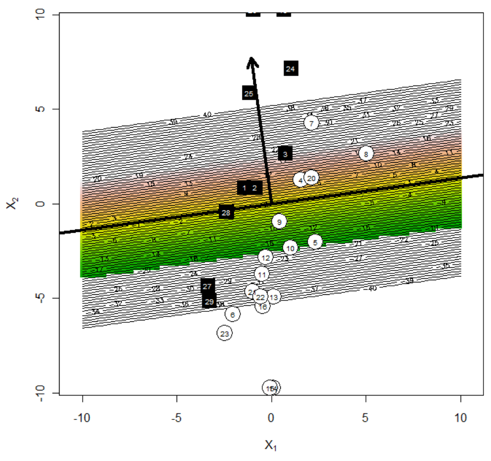
   ```
   data = load.data()
   plot(data)
   m1 = colMeans(data$X[data$y==1,])
   m0 = colMeans(data$X[data$y==-1,])
   mark.point(m1)
   mark.point(m0, -1)
   ```



2. The code:

```
w = m1 - m0
plot.classifier(w)
plot.data(data, add=T)
```

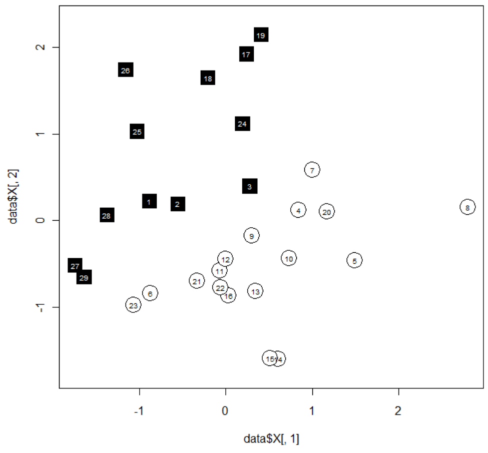
Produces the following image:



Seven points seem to be misclassified here.

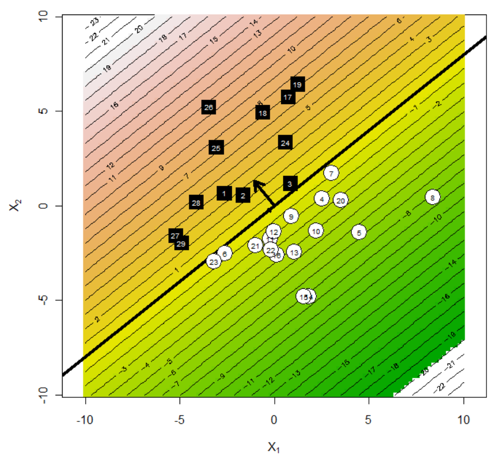
3. Covariance normalization of the data:

```
sigma = cov(data$X)
normalizer = solve(matrix_sqrt(sigma))
new_data = data
new_data$X = new_data$X %*% normalizer
plot(new_data)
```



4. Now we build a “difference-of-means” classifier on the transformed data:

```
new_m1 = colMeans(new_data$X[new_data$y==1,])
new_m0 = colMeans(new_data$X[new_data$y==-1,])
new_w = new_m1 - new_m0
plot.classifier(new_w)
new_data$X = 3*new_data$X # Just to zoom in the data
plot(new_data, add=T)
```



It seems to classify the data perfectly.

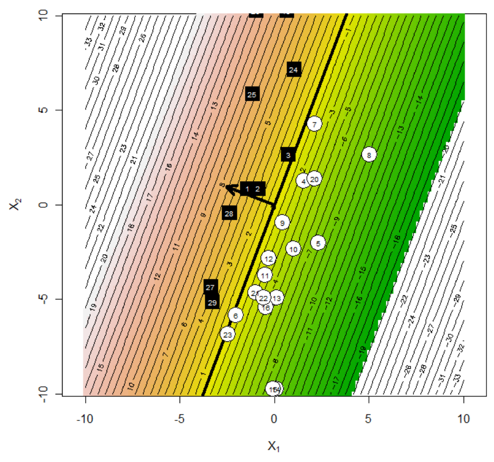
5. If we compute the sigma differently:

```
sigma1 = cov(data$X[data$y==1,])
sigma2 = cov(data$X[data$y==-1,])
sigma = (sigma1 + sigma2)/2
normalizer = solve(matrix_sqrt(sigma))
new_data = data
new_data$X = new_data$X %*% normalizer
...
```

The resulting classifier, in our case, turns out to be not much different from the previous one.

6. The proper way of computing the Fisher's discriminant is the following:

```
sigma1 = cov(data$X[data$y==1,])
sigma2 = cov(data$X[data$y==-1,])
sigma = (sigma1 + sigma2)/2
m1 = colMeans(data$X[data$y==1,])
m0 = colMeans(data$X[data$y==-1,])
w_fda = solve(sigma) %*% (m1 - m0)
plot.classifier(w_fda)
plot(data, add=T)
```



7. This should be familiar from Exercise 3:

```
w0_fda = -t(w_fda) %*% (m1 + m0)/2
plot.classifier(w_fda, w0_fda)
plot(data, add=T)
mark.point((m1 + m0)/2)
```

   Note that the resulting classifier in our case turned out to be worse than the one with $w_0 = 0$.

8.
```
> library(MASS)
> w_lda = lda(data$X, data$y)$scaling

> w_fda / w_lda
       [,1]
x1 3.280373
x2 3.114978
```

   Note that the two vectors are not exactly proportional: the implementation of the `lda` function differs in how the covariance matrix is estimated (it uses `cov(data$X)`, it seems).

Least-squares Classifier

Least-squares classifier

Exercise 10 (1pt). Define

```
n1 = sum(data$y == 1)
n0 = sum(data$y == -1)
```

Now define a vector $\mathbf{y}'$ such that:

$$y'_i = \begin{cases} \frac{1}{n_1} & \text{if } y_i = 1 \\ -\frac{1}{n_0} & \text{otherwise.} \end{cases}$$

Finally, compute $\mathbf{w}$ using the least squares rule:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}'.$$

Plot the resulting classifier. Verify that the resulting weight vector is equal, up to a constant, to the Fisher discriminant with $\boldsymbol{\Sigma}$ estimated as `cov(data$X)`.

If you want to know why and when this happens, meditate at the two equations below for a minute and you shall see:

$$\text{Fisher's discriminant: } \mathbf{w} = (\quad \boldsymbol{\Sigma} \quad)^{-1}(\mathbf{m}_1 - \mathbf{m}_0)$$
$$\text{Least-squares: } \mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}(\quad \mathbf{X}^T\mathbf{y}' \quad)$$

Solution.

```
n1 = sum(data$y == 1)
n0 = sum(data$y == -1)

yy = data$y
yy[which(yy == 1)] = yy[which(yy == 1)] / n1
yy[which(yy == -1)] = yy[which(yy == -1)] / n0

X=data$X
w_ls = solve(t(X) %*% X) %*% t(X) %*% yy
```

Re-using the `m0` and `m1` variables from the previous exercise, let us compute the Fisher's discriminant with $\hat{\boldsymbol{\Sigma}}$ =`cov(data$X)` and compare it to `w_ls`:

```
> w_fda0 = solve(cov(data$X)) %*% (m1 - m0)
> w_fda0 / w_ls
    [,1]
x1   28
x2   28
```

We see that the weight vectors are indeed proportional to each other, with the coefficient of proportionality equal to $n - 1$. Coincidence?

Perceptron

Exercise 11-12 (2pt). The perceptron algorithm is based on batch or on-line gradient optimization of the error function:

Batch perceptron

$$\mathcal{E}(\mathbf{w}) = \sum_{\mathbf{x}_i \text{ is misclassified}} -\mathbf{w}^T\mathbf{x}_i y_i$$

The expression $\mathbf{w}^T\mathbf{x}_i y_i$ is called the *functional margin* of the training example $\mathbf{x}_i$.

1. Prove that a training example is misclassified *iff* its functional margin is negative.

2. Implement the function `df(w, data)` that computes the gradient of $\mathcal{E}$.

3. Start with `w = c(0, 0)`. Update $\mathbf{w}$ using a single step of the perceptron algorithm: $\mathbf{w} = \mathbf{w} - \mu\,\mathrm{df}(\mathbf{w}, \mathrm{data})$ and plot the classifier with the data (using `plot.classifier` and `plot.data`). Repeat the step and plot again. Repeat iterating and plotting until the algorithm converges to a solution. How many steps did it take?

   Hint: Use `par(mfrow=c(3,3))` to put multiple plots on a single figure.

4. Try changing μ. Does anything change besides the scale?

5. Optional bonus: try using the `animation` package as follows:

```r
library(animation)
ani.options(interval=0.2)
ani.start()
# ... Algorithm loop which outputs a number of plots
ani.stop()
```

Solution.

1. A training example $\mathbf{x}_i$ is misclassified iff $\mathrm{sign}(f_\mathbf{w}(\mathbf{x}_i)) \neq y_i$. But then

$$\mathrm{sign}(f_\mathbf{w}(\mathbf{x}_i)y_i) = -1,$$
$$\mathrm{sign}(\mathbf{w}^T\mathbf{x}_i y_i) = -1,$$

   i.e. the functional margin must indeed be negative.

2. The gradient of $\mathcal{E}$ is:

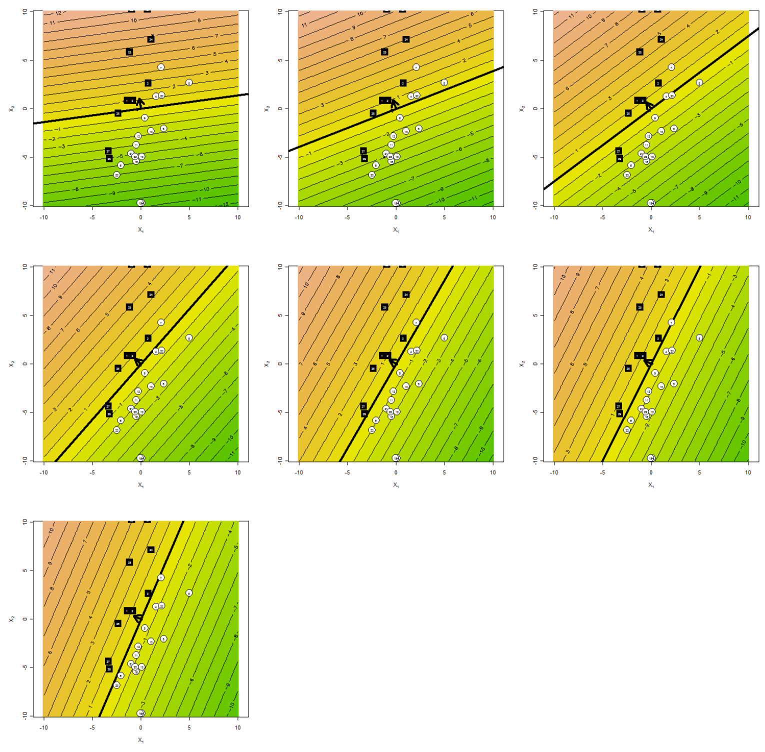
$$\nabla\mathcal{E}(\mathbf{w}) = \sum_{\mathbf{x}_i \text{ is misclassified}} -y_i\mathbf{x}_i.$$

   Note that despite the lack of explicit $\mathbf{w}$ on the right side, $\mathcal{E}$ does depend on $\mathbf{w}$ (think why).

A possible implementation is, for example, the following:

```r
df = function(w, data) {
  margins = (data$X %*% w) * data$y
  r = c(0, 0)
  for (i in which(margins <= 0)) {
    r = r - data$X[i,]*data$y[i]
  }
  r
}
```

3.
```r
par(mfrow = c(3,3))
w = c(0, 0)
while(df(w, data) != c(0, 0)) {
  w = w - 0.01*df(w, data)
  plot.classifier(w)
  plot(data, add=T)
}
```



The algorithm converged in 7 steps.

4. By trying different values of μ it is easy to note that it influences nothing besides the scale of the $\mathbf{w}$ vector. The algorithm always performs the same 7 steps.

Exercise 13 (1pt). Implement the on-line perceptron algorithm and visualize its convergence as in the previous exercise. The algorithm selects a single misclassified example on each step. Try highlighting this example on each iteration's plot using `mark.point`.

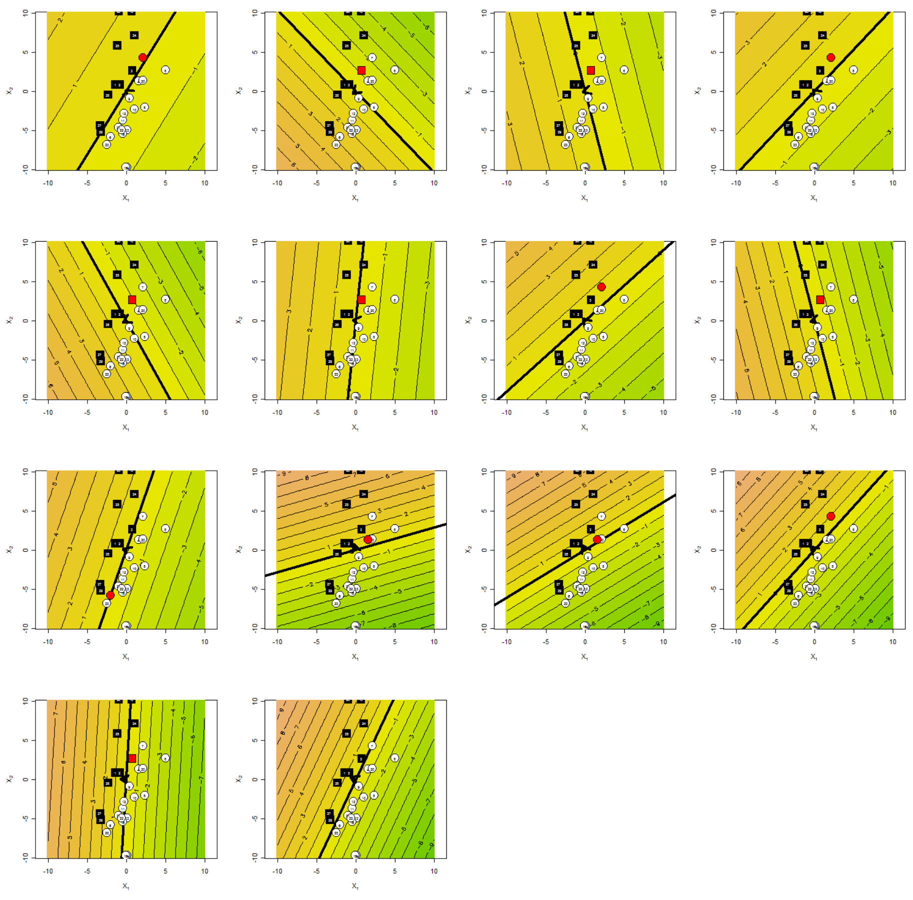
Online perceptron

Solution. Here is the solution with the `animation` package. It takes 14 steps to converge.

```
find.misclassified = function(w, data) {
  margins = (data$X %*% w) * data$y
  which(margins <= 0)[1]
}

library(animation)
ani.options(interval=0.2)
ani.start()
w = c(0, 0)
while(!is.na(find.misclassified(w, data))) {
  i = find.misclassified(w, data)
  if (w != c(0, 0)) mark.point(data$X[i,], data$y[i])
  w = w + 0.1*data$y[i]*data$X[i,]
  plot.classifier(w)
  plot(d, add=T)
}
ani.stop()
```

Note that the choice of μ is made somewhat arbitrarily. The value of 0.1 seemed to produce nicer pictures (too large μ results in ugly dense level lines).



Exercise 14 (1pt). In two previous exercises you implemented the perceptron algorithm without the bias term w_0. Modify the algorithms to also search for w_0. Note that you may do it implicitly by simply adding a column of ones to the data matrix. However here I ask you to do it explicitly (in addition, this will integrate nicely with the current plotting logic). *Perceptron with bias*

Solution. The objective function which includes w_0 is:

$$\mathcal{E}(\mathbf{w}, w_0) = \sum_{\mathbf{x}_i \text{ is misclassified}} -(\mathbf{w}^T\mathbf{x}_i + w_0)y_i.$$

The gradient of $\mathcal{E}$ with respect to w_0 is then:

$$\nabla_{w_0}\mathcal{E}(\mathbf{w}, w_0) = \sum_{\mathbf{x}_i \text{ is misclassified}} -y_i.$$

Consequently, the updated on-line perceptron algorithm is:

```r
find.misclassified = function(w, w0, data) {
  margins = (data$X %*% w + w0) * data$y
  which(margins <= 0)[1]
}

w = c(0, 0)
w0 = 0
while(!is.na(find.misclassified(w, w0, data))) {
  i = find.misclassified(w, w0, data)
  if (w != c(0, 0)) mark.point(data$X[i,], data$y[i])
  w = w + data$y[i]*data$X[i,]
  w0 = w0 + data$y[i]
  plot.classifier(w, w0)
  plot(d, add=T)
}
```

The modification to batch perceptron is similar.

Exercise 15 (1pt). By now you should have implemented the *Fisher's discriminant*, the *Least squares algorithm*, the *Batch perceptron* and *Online perceptron* algorithms, each of them found its own weight vector. Now use *Logistic regression* (the `glm` function, see lecture slides) and *SVM* (the `svm` function from the `e1071` package, see lecture slides). Verify that the weights found by all algorithms are similar up to a constant.

Library functions

Hint: For the `svm` model use `scale=F`. You can then recover the weight vector and bias term as follows:

```r
w_svm = t(m$coefs) %*% m$SV
w0_svm = -m$rho
```

Solution. The following code shows how to invoke the logistic regression and SVM classifier learning algorithms from R:

```r
# Logistic regression
yy = (data$y + 1)/2    # {-1, 1} -> {0, 1}
m = glm(yy ~ data$X, family=binomial)
w_glm = m$coefficients[2:3]
w0_glm = m$coefficients[1]

# SVM
library('e1071')
m = svm(data$X, data$y, kernel='linear', scale=F)
w_svm = t(m$coefs) %*% m$SV
w0_svm = -m$rho
```
