Linear Classification

1. Binary classification on Spam Dataset

We illustrate the task of binary classification on the spam dataset from the UCI machine learning repository (Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science).

Short description of the spam dataset, as it appears on the repository: *The "spam" concept is diverse: advertisements for products/web sites, make money fast schemes, chain letters, pornography... Our collection of spam e-mails came from our postmaster and individuals who had filed spam. Our collection of non-spam e-mails came from filed work and personal e-mails, and hence the word 'george' and the area code '650' are indicators of non-spam. These are useful when constructing a personalized spam filter. One would either have to blind such non-spam indicators or get a very wide collection of non-spam to generate a general purpose spam filter. The last column of 'spambase.data' denotes whether the e-mail was considered spam (1) or not (0), i.e. unsolicited commercial e-mail. Most of the attributes indicate whether a particular word or character was frequently occuring in the e-mail. The run-length attributes (55-57) measure the length of sequences of consecutive capital letters.*

Load and parse the data as follows.

```
library(RCurl)
```

```
## Loading required package: bitops
```

```
downloaded.url <-
  getURL('https://archive.ics.uci.edu/ml/machine-learning-databases/spambase/spambase.data')
connection <- textConnection(downloaded.url)
Spam <- read.csv(connection, header=FALSE)
str(Spam)
```

```
## 'data.frame':    4601 obs. of  58 variables:
##  $ V1 : num  0 0.21 0.06 0 0 0 0 0 0.15 0.06 ...
##  $ V2 : num  0.64 0.28 0 0 0 0 0 0 0 0.12 ...
##  $ V3 : num  0.64 0.5 0.71 0 0 0 0 0 0.46 0.77 ...
##  $ V4 : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V5 : num  0.32 0.14 1.23 0.63 0.63 1.85 1.92 1.88 0.61 0.19 ...
##  $ V6 : num  0 0.28 0.19 0 0 0 0 0 0 0.32 ...
##  $ V7 : num  0 0.21 0.19 0.31 0.31 0 0 0 0.3 0.38 ...
##  $ V8 : num  0 0.07 0.12 0.63 0.63 1.85 0 1.88 0 0 ...
##  $ V9 : num  0 0 0.64 0.31 0.31 0 0 0 0.92 0.06 ...
##  $ V10: num  0 0.94 0.25 0.63 0.63 0 0.64 0 0.76 0 ...
##  $ V11: num  0 0.21 0.38 0.31 0.31 0 0.96 0 0.76 0 ...
##  $ V12: num  0.64 0.79 0.45 0.31 0.31 0 1.28 0 0.92 0.64 ...
##  $ V13: num  0 0.65 0.12 0.31 0.31 0 0 0 0 0.25 ...
##  $ V14: num  0 0.21 0 0 0 0 0 0 0 0 ...
##  $ V15: num  0 0.14 1.75 0 0 0 0 0 0 0.12 ...
##  $ V16: num  0.32 0.14 0.06 0.31 0.31 0 0.96 0 0 0 ...
##  $ V17: num  0 0.07 0.06 0 0 0 0 0 0 0 ...
##  $ V18: num  1.29 0.28 1.03 0 0 0 0.32 0 0.15 0.12 ...
##  $ V19: num  1.93 3.47 1.36 3.18 3.18 0 3.85 0 1.23 1.67 ...
##  $ V20: num  0 0 0.32 0 0 0 0 0 3.53 0.06 ...
```

```
##  $ V21: num  0.96 1.59 0.51 0.31 0.31 0 0.64 0 2 0.71 ...
##  $ V22: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V23: num  0 0.43 1.16 0 0 0 0 0 0 0.19 ...
##  $ V24: num  0 0.43 0.06 0 0 0 0 0 0.15 0 ...
##  $ V25: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V26: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V27: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V28: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V29: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V30: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V31: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V32: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V33: num  0 0 0 0 0 0 0 0 0.15 0 ...
##  $ V34: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V35: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V36: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V37: num  0 0.07 0 0 0 0 0 0 0 0 ...
##  $ V38: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V39: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V40: num  0 0 0.06 0 0 0 0 0 0 0 ...
##  $ V41: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V42: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V43: num  0 0 0.12 0 0 0 0 0 0.3 0 ...
##  $ V44: num  0 0 0 0 0 0 0 0 0 0.06 ...
##  $ V45: num  0 0 0.06 0 0 0 0 0 0 0 ...
##  $ V46: num  0 0 0.06 0 0 0 0 0 0 0 ...
##  $ V47: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V48: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V49: num  0 0 0.01 0 0 0 0 0 0 0.04 ...
##  $ V50: num  0 0.132 0.143 0.137 0.135 0.223 0.054 0.206 0.271 0.03 ...
##  $ V51: num  0 0 0 0 0 0 0 0 0 0 ...
##  $ V52: num  0.778 0.372 0.276 0.137 0.135 0 0.164 0 0.181 0.244 ...
##  $ V53: num  0 0.18 0.184 0 0 0 0.054 0 0.203 0.081 ...
##  $ V54: num  0 0.048 0.01 0 0 0 0 0 0.022 0 ...
##  $ V55: num  3.76 5.11 9.82 3.54 3.54 ...
##  $ V56: int  61 101 485 40 40 15 4 11 445 43 ...
##  $ V57: int  278 1028 2259 191 191 54 112 49 1257 749 ...
##  $ V58: int  1 1 1 1 1 1 1 1 1 1 ...
```

```
attach(Spam)
n = nrow(Spam)
```

The data contains 57 attributes, for 4601 observations. We predict the label of an email (Spam or Non-Spam / 1 or 0 / corresponding to the last column, denoted V58) using all predictors, with a logistic regression model first.

The logistic regression model is fitted using the command `glm` (aka *Generalised Linear Model*). We must use `family=binomial` in its arguments to specify the link function to be used in the model; here a logistic link. The coefficients are extracted using the function `coef`. To make predictions using the fitted model, we use the command `predict`. To specify R that we want to return the class conditional probabilities $P(Y = 1|X = x)$, we must put `type = "response"` in its arguments.

To test the prediction accuracy of the logistic regression model, we divide randomly the dataset into two subsets: train and test. We use 3000 observations to train the model, and the remaining 1600 to assess its performance.

```
n.train = 3000
n.test = n-n.train
train = sample(n, n.train)
test = setdiff(1:n, train)
test.data.x = Spam[test, -ncol(Spam)]
test.data.y = Spam[test, ncol(Spam)]

lr.fit = glm(V58~., data=Spam, family=binomial, subset=train)
lr.probs=predict(lr.fit, test.data.x, type="response")

lr.pred = rep(0, n.test)
lr.pred[lr.probs >.5] = 1
table(lr.pred, test.data.y)
```

```
##         test.data.y
## lr.pred   0   1
##       0 896  67
##       1  44 594
```

The test error is:

```
1-mean(lr.pred == test.data.y)
```

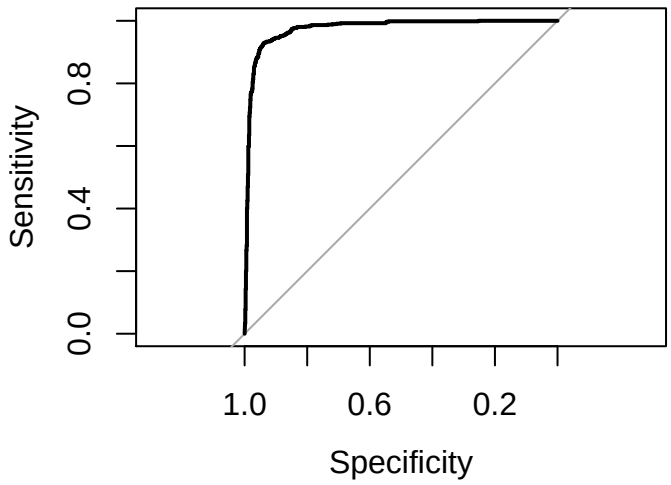
```
## [1] 0.06933167
```

In addition, we plot a ROC curve to assess the test error for different threshold values t: classify as spam if the predicted probability is larger than t, and as non-spam otherwise. Let: TP = the number of true positives, TN = the number of true negatives, FP = the number of false positives, and FN = the number of false hegatives. A ROC curve plots the sensitivity (TP/(TP+FN)) as a function of the specificity (TN/(TN+FP)), see chapter SL: FOUNDATIONS. We use the function `roc`, which is part of the package `pROC`. The Area Under the Curve (AUC) is returned using `auc = TRUE`.

```
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
##
## Attaching package: 'pROC'
##
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
roc(test.data.y, lr.probs, auc=TRUE, plot=TRUE)
```




```
##
## Call:
## roc.default(response = test.data.y, predictor = lr.probs, auc = TRUE,     plot = TRUE)
##
## Data: lr.probs in 940 controls (test.data.y 0) < 661 cases (test.data.y 1).
## Area under the curve: 0.9746
```

We perform the analysis again using LDA.

```
library(MASS)
lda.fit = lda(V58~., data=Spam, subset=train)
lda.pred = predict(lda.fit, test.data.x)
names(lda.pred)
```

```
## [1] "class"     "posterior" "x"
```

```
lda.class = lda.pred$class
table(lda.class, test.data.y)
```

```
##          test.data.y
## lda.class   0   1
##         0 902 132
##         1  38 529
```

```
1-mean(lda.class == test.data.y)
```

```
## [1] 0.1061836
```

LDA performs worse than the logistic model on this dataset.

2. Multiclass classification on MNIST Dataset

Load the MNIST digit recognition training dataset into R from Kaggle https://www.kaggle.com/c/digit-recognizer/data

Each image is 28 by 28 pixels for a total of $d = 784$ pixels. Each pixel value is an integer between 0 and 255. We make use of the training dataset only, called 'train.csv', which has 785 columns, the first column being the label of the image.

```
data <- read.csv("train.csv")
head(names(data))
```

```
## [1] "label"  "pixel0" "pixel1" "pixel2" "pixel3" "pixel4"
```

We divide the dataset into training and test examples. Two thirds of the original data are retained for training (corresponding to 28000 observations), and the remaining third to check the model predictive accuracy (14000 observations).

```
n = nrow(data)
train = 1:(2*n/3)
train.X = data[train,-1]
train.Y = data[train, 1]
test.X  = data[-train,-1]
test.Y  = data[-train, 1]
```

We reduce the dimensionality of the input space by keeping the principal components accounting for most of the variability in the data. An overview of PCA and its implementation can be found in the chapter UL: PRINCIPAL COMPONENT ANALYSIS.

```
pc.train = prcomp(train.X)
pc.train.X = pc.train$x
pc.var = pc.train$sdev^2
pve = pc.var/sum(pc.var)
cumsum(pve)[80]
```

```
## [1] 0.8910555
```

We observe that the first 80 components explain about 90% of the variance in the data. The principal components calculated on the training data are used to project the test data.

```
pc.test.X = predict(pc.train, newdata = test.X)
```

With the pre-processed data `pc.train.X` and `pc.test.X` at hand, we test the logistic regression, LDA and QDA models on the first 80 principal components. For multivariate logistic regression, we use the function `multinom` from the package `nnet`. We use the argument `"class"` in `predict` to indicate that we are interested in the final classification result. If one is interested in the class conditional probabilities, we must put `"probs"`.

```
train.data = data.frame(cbind(train.Y, pc.train.X[,1:80]))

library(nnet)
lr.fit = multinom(train.Y~., data = train.data)
```

```
## # weights:  820 (729 variable)
## initial  value 64472.382604
## iter  10 value 12598.427421
## iter  20 value 11945.954268
```

```
## iter  30 value 11851.976385
## iter  40 value 11827.130885
## iter  50 value 11819.976844
## iter  60 value 11818.273682
## iter  70 value 11817.781694
## iter  80 value 11817.318624
## iter  90 value 11816.441876
## iter 100 value 11812.742751
## final  value 11812.742751
## stopped after 100 iterations
```

```
lr.pred = predict(lr.fit, newdata = data.frame(pc.test.X[,1:80]), "class")
table(lr.pred, test.Y)
```

```
##         test.Y
## lr.pred    0    1    2    3    4    5    6    7    8    9
##       0 1052    0    9    2    4   20   11    1    3    8
##       1    1 1531   22   15   22   46    5   23   67   22
##       2   21    7 1166   47    6   11   15   19   20    3
##       3   11    8   21 1255    1   75    0    2   55   23
##       4   18    0   13    2 1198   20   12   20   15   84
##       5  210    9    8   47    1 1010   18    1   35    8
##       6   46    3   39   12   33   37 1294    1   17    3
##       7   27    2   30   20    5   14    0 1380   13  115
##       8   20   16   37   27    8   22    3    3 1118   10
##       9    5    1    8   15   38   19    0   27   26 1147
```

```
1-mean(lr.pred == test.Y)
```

```
## [1] 0.1320714
```

```
lda.fit = lda(train.Y~., data = train.data)
lda.pred = predict(lda.fit, newdata = data.frame(pc.test.X[,1:80]))
lda.class = lda.pred$class
table(lda.class, test.Y)
```

```
##          test.Y
## lda.class    0    1    2    3    4    5    6    7    8    9
##         0 1322    0   10    3    3   22   18    7    6   10
##         1    0 1521   38   21   18   16   15   29   89   12
##         2    2    5 1104   50    9    3   11   12    8    3
##         3    5    2   34 1206    0   64    1    3   37   23
##         4    5    2   27    2 1163   23   19   27   12   84
##         5   38   10   14   54    6 1011   32    2   59   11
##         6   19    1   24   12   13   27 1252    0   13    2
##         7    3    3   22   28    2   11    0 1286    5   46
##         8   14   32   67   35    9   71    9    5 1104    9
##         9    3    1   13   31   93   26    1  106   36 1223
```

```
1-mean(lda.class == test.Y)
```

```
## [1] 0.1291429
```

```
qda.fit = qda(train.Y~., data = train.data)
qda.pred = predict(qda.fit, newdata = data.frame(pc.test.X[,1:80]))
qda.class = qda.pred$class
table(qda.class, test.Y)
```

```
##          test.Y
## qda.class    0    1    2    3    4    5    6    7    8    9
##         0 1389    0    4    0    2    6   11    2    3    4
##         1    0 1520    0    1    1    0    1    0    3    0
##         2    6   20 1314   21    5    4    2   19   10    7
##         3    2    1    7 1370    0   19    1    2   16   14
##         4    0    4    1    3 1277    0    1    8    2   23
##         5    3    0    0   11    0 1221   28    1    5    4
##         6    1    1    2    0    2    3 1307    0    1    0
##         7    0    2    2    2    2    0    0 1399    2   22
##         8   10   29   23   31    7   19    7   15 1322   30
##         9    0    0    0    3   20    2    0   31    5 1319
```

```
1-mean(qda.class == test.Y)
```

```
## [1] 0.04014286
```

The QDA approach performs the best here, with an error rate around 4%.