Oblig 2

Håkon Kvernmoen

3/30/2021

Problem 1

a)

We begin by loading the data and saving it to the `spam` data-frame. We also include the `MASS` library to use GLM's

```
fil = "https://www.uio.no/studier/emner/matnat/math/STK2100/data/spam_data.txt"
spam = read.table(fil, header=T)
library(MASS)
```

We now preform a logistic regression on the training data, using all the explanatory variables. The `train` column only indicates if the data point should be used for training, so we do not include this variable in our fit, but rather use it as a subset to preform the fit.

```
fit = glm(y~.-train, data=spam, subset=spam$train, family = binomial)
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
pred = predict(fit, spam[!spam$train,], type="response")>0.5
```

The most natural measure to use for prediction performance is to compute the confusion matrix. This is a 2×2 matrix with TN (true negative) and TP (true positive) on the diagonal. The element below the diagonal is FN (false negative) and above the diagonal FP (false positive). To make our measure more concise and direct we will calculate the TPR (true positive rate) and TNR (true negative rate). These are defined as

$$\text{Confusion matrix: } \begin{pmatrix} TN & FN \\ FP & TP \end{pmatrix} \quad TPR = \frac{TP}{TP+FN} \quad \text{and} \quad TNR = \frac{TN}{TN+FP}$$

We aim to have both TPR and TNR close to one. If TPR = 1 we have no false negatives and if TNR = 1 we have no false positives. If FN = TP and TN = FP we have TPR = 0.5 and TNR = 0.5 and the models preforms no better than guessing. We compute the confusion matrix and make a function to calculate TPR/TNR as we will use them a lot.

```
fit.confusion_mat = table(pred, spam$y[!spam$train])

set_dec = function(x, k) trimws(format(round(x, k), nsmall=k))
true_PN_rate = function (mat, dec=4, display=F) {
```

```
  TPR = mat[2,2]/(mat[2,2]+mat[1,2])
  TNR = mat[1,1]/(mat[1,1]+mat[2,1])

  if(display) {
    paste("TPR = ", set_dec(TPR, dec), "TNR = ", set_dec(TNR, dec))
  }
  else {
    c(TPR, TNR)
  }
}

fit.confusion_mat # Print confusion matrix
```

```
##
## pred    FALSE TRUE
##   FALSE  1733  148
##   TRUE    114 1070
```

```
print(true_PN_rate(fit.confusion_mat, display=T))
```

```
## [1] "TPR =  0.8785 TNR =  0.9383"
```

Thus $TPR \approx 0.88$ and $TNR \approx 0.94$. How accurate it needs to be is dependent on the application, but we can probably obtain better results using a more complex model. The goal of a spam filter is to hide unwanted emails from the users inbox. Two or three spam emails slipping trough is not the end of the world, but we should defiantly try to avoid marking an actual email as spam (FP) since it migth contain important information such as a change in flight times or your rent bill. Thus the most important measure for our purposes is to maximize the TNR.

b)

We have $p = 57$ explanatory variables. To reduce the dimensionality of the data while keeping as much of the original relationships as possible, we compute the principle components.

```
x.prcomp = prcomp(spam[,1:57], retx=T, scale=T)
d = data.frame(x.prcomp$x, y=spam$y, train=spam$train)
```

The parameter `scale` is set equal to true since we have not scaled the explanatory variables (design matrix) before computing the principle components. Scaling is important, since the PCA relies on finding the linear combinations of the columns from the design matrix that has maximum variance. If lets say the first explanatory variable (first column of design matrix) is 10 times bigger than all the other explanatory variables it would be over-represented in the PCA since the variance is higher (even tough the "spread" might be close to equal). When preforming a PCA we are only interested in the "spread" of each explanatory variable, independent of the data being on the interval $[-5, 5]$ or $[-5000, 5000]$. Thus scaling our explanatory variables such that the variance is a true measure of the actual "spread" of the data is crucial.

We then preform the logistic regression using the 2 first principal components.

```
fit.pc = glm(y~.-train, data=d[, c(1:2, 58,59)], family = binomial, subset=d$train)
pred.pc = predict(fit.pc, d[!d$train, ], type="response")>0.5
fit.pc.confusion_mat = table(pred.pc, d$y[!d$train])

fit.pc.confusion_mat
```

```
##
## pred.pc FALSE TRUE
##   FALSE  1722  279
##   TRUE    125  939
```

```
print(true_PN_rate(fit.pc.confusion_mat, display=T))
```

```
## [1] "TPR =  0.7709 TNR =  0.9323"
```

We observe that FN increases quite a lot an thus we end up with $TPR \approx 0.77$. FP (the one we are most worried about) increases just slightly resulting in $TNR \approx 0.93$. These are the most important components, but does not necessarily contain as much of the relationship between the explanatory variables and the target as all the 57 "raw" explanatory variables. This can be seen for instance by printing out the 5 largest principal components (square root of the eigenvalues of the covariance matrix)

```
x.prcomp$sdev[1:5]
```

```
## [1] 2.567476 1.807602 1.415327 1.270102 1.243466
```

As we can see the to first principal components are largest (2.56... and 1.80...) but the rest are not dramatically smaller than these two. This is a pointer for us to include more than just two principal components.

c)

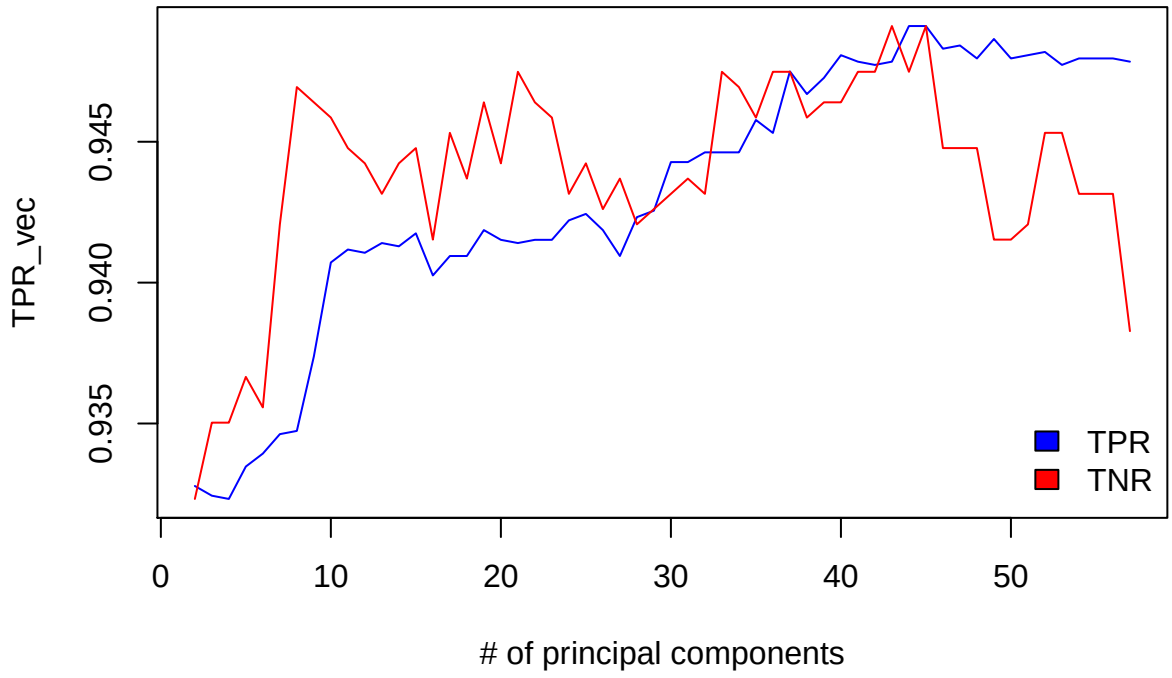
We will now try to increase the number of principal components in our logistic regression model. As a measure of prediction performance we will use the same FP/FN percentages as in **a** and **b**.

```
min_pc = 2
max_pc = 57
n_pc = max_pc-min_pc+1
n_pc_vec = seq(min_pc, max_pc, length.out = n_pc)
TPR_vec = seq(0, 0, length.out = n_pc)
TNR_vec = seq(0, 0, length.out = n_pc)
var_sum_vec = seq(0, 0, length.out = n_pc)

for( i in 1:n_pc) {
  k = i+1
  fit.temp = glm(y~.-train, data=d[, c(1:k, 58,59)], family = binomial, subset=d$train)
  pred.temp = predict(fit.temp, d[!d$train, ], type="response")>0.5
  fit.temp.confusion_mat = table(pred.temp, d$y[!d$train])
  TR = true_PN_rate(fit.temp.confusion_mat)
  TPR_vec[i] = TR[1]
```

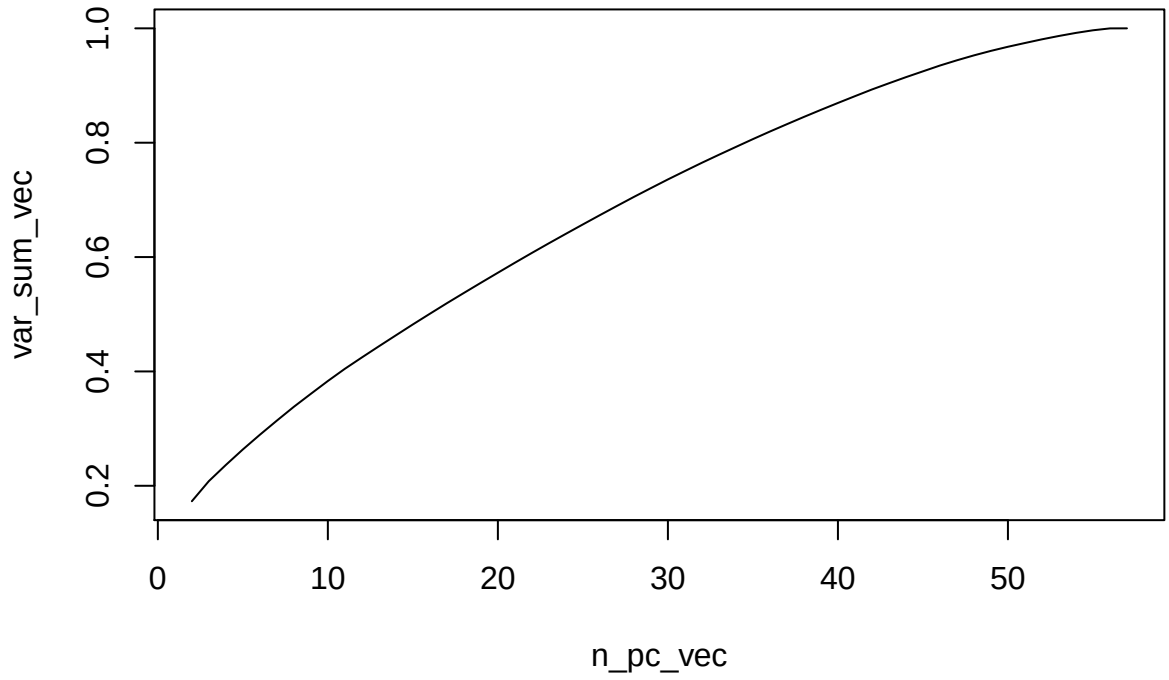
```
  TNR_vec[i] = TR[2]
  var_sum_vec[i] = sum(x.prcomp$sdev[1:k]^2)
}

var_sum_vec = var_sum_vec/sum(x.prcomp$sdev^2)
cols = c("blue", "red")
plot(n_pc_vec, TPR_vec, type="l", col=cols[1], axes=FALSE, xlab="# of principal components")
par(new=TRUE)
plot(n_pc_vec, TNR_vec, type="l", col=cols[2], xlab="", ylab="")
legend("bottomright", c("TPR","TNR"), fill=cols, bty="n")
```



How many to use? Dependes, reason is to decrease dimensionality so should prob use less. cumulative sum?

```
plot(n_pc_vec, var_sum_vec, type="l")
```



d)

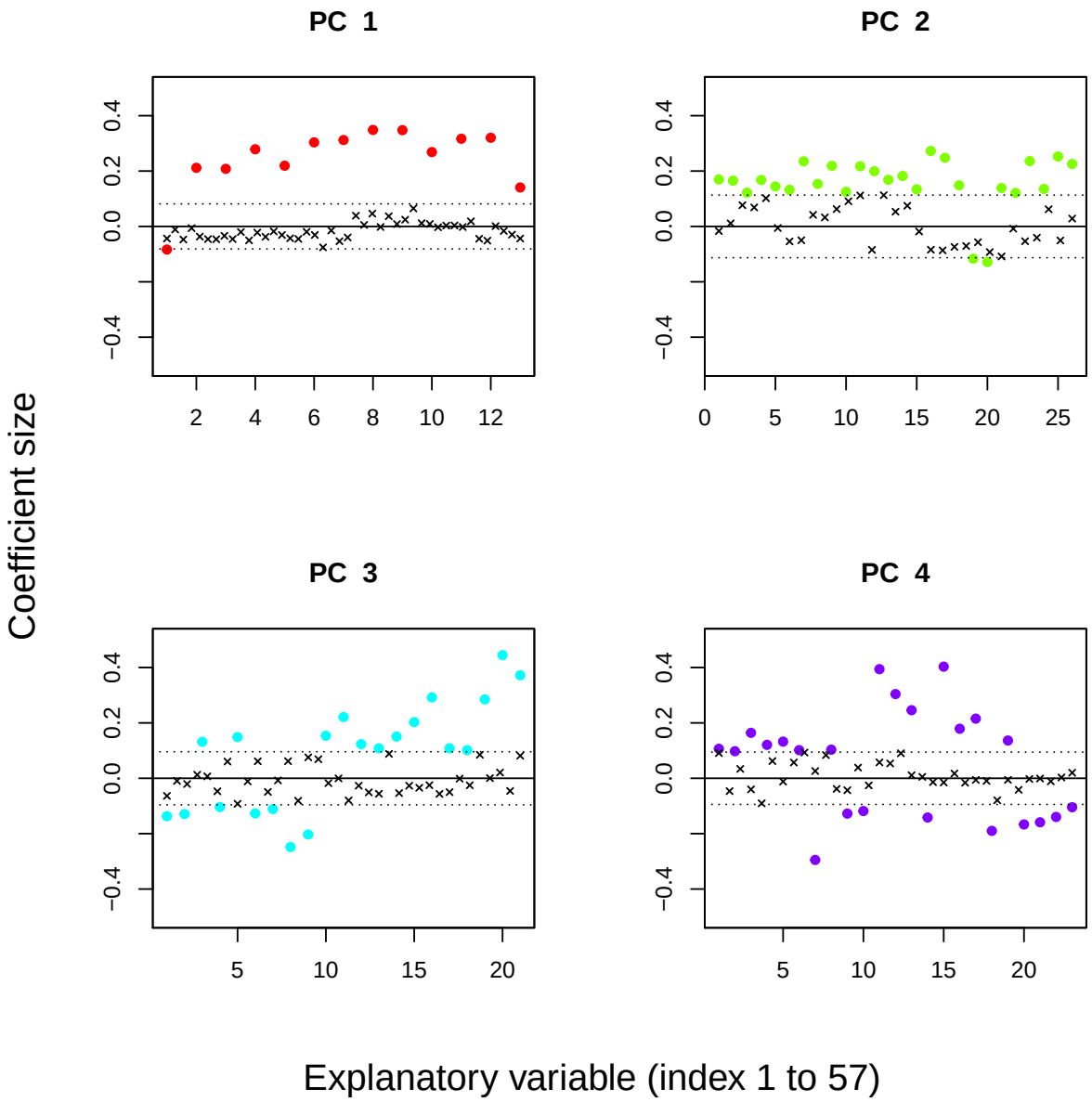
When using the most important principal components we are granted with new variables that explain more of the data than the original variables. These new variables are linear combinations of the original variables, so the trade off is that we loose some interpretation of the model. Some analysis of the principal components is thus desired to achieve better interpretability. Since we use quite a lot of principal components, deep diving into the structure of each of these will be time consuming so we will here focus on the 4 most important (with the largest eigenvalue).

Each principal component will have 57 coefficients corresponding to the 57 explanatory variables. Some will be large (meaning this variable is important for this principal component) and some small or (very close to) zero (meaning this variable is **not** important to this principal component). To pick out the "large" components we plot for each principal component the coefficients that are larger than the mean of the absolute value of all the coefficients for that principal component. All other coefficients are "small" and are plotted as crosses, while the "large" ones are plotted as colored circles. The solid line is zero while the two dotted lines are $\pm$ the mean of the absolute value of the coefficients.

```
n = 4
cols = rainbow(n)
par(mfrow=c(2,2), oma=c(2,2,0,0))
for( i in 1:n ) {
  avrg_coef_size = mean(abs(x.prcomp$rotation[,i]))
  pc.eig= x.prcomp$rotation[,i]
  pc.eig.large = pc.eig[abs(pc.eig) > avrg_coef_size]
  pc.eig.small = pc.eig[abs(pc.eig) < avrg_coef_size]
```

```
  plot.ts(pc.eig.large, type="p", col=cols[i], ylab="", main=paste("PC ", i), xlab="", pch=19, ylim=c(-0
  par(new=T)
  plot.ts(pc.eig.small, type="p",pch=4, cex=0.6,  yaxt="n", xaxt="n",ann=F, ylim=c(-0.5,0.5))
  abline(0,0)
  abline(avrg_coef_size, 0, lty="dotted")
  abline(-avrg_coef_size, 0, lty="dotted")
}

mtext("Explanatory variable (index 1 to 57)", side=1, line=0, outer=T, cex=1.3)
mtext("Coefficient size", side=2, line=0, cex=1.3, outer=T)
```



To interpret this we quickly summarize what each explanatory variable means.

e)

We now try some non-linear models. First we fit the model using the 3 first explanatory variables using smoothing splines with 4 degrees of freedom. Then we use the same 3 explanatory variables to fit a logistic regression model. We can then check if these non-linear terms yields any improvements.

```
library(gam)
```

```
## Loading required package: splines
```

```
## Loading required package: foreach
```

```
## Loaded gam 1.20
```

```
fit.gam = gam(y~s(x1)+s(x2)+s(x3), data=spam, subset =spam$train, family = binomial) # Smoothing spline
fit.log = gam(y~x1+x2+x3, data=spam, subset=spam$train, family = binomial) # Logistic regression

pred.gam = predict(fit.gam, spam[!spam$train, ], type="response") > 0.5
pred.log = predict(fit.log, spam[!spam$train, ], type="response") > 0.5

confusion.gam = table(pred.gam, d$y[!d$train])
confusion.log = table(pred.log, d$y[!d$train])

print(paste("Natural Splines -> ", true_PN_rate(confusion.gam, display = T)))
```

```
## [1] "Natural Splines ->  TPR =  0.4729 TNR =  0.8500"
```

```
confusion.gam
```

```
##
## pred.gam FALSE TRUE
##    FALSE  1570  642
##    TRUE    277  576
```

```
print(paste("Logistic regression -> ", true_PN_rate(confusion.log, display = T)))
```

```
## [1] "Logistic regression ->  TPR =  0.2233 TNR =  0.9123"
```

```
confusion.log
```

```
##
## pred.log FALSE TRUE
##    FALSE  1685  946
##    TRUE    162  272
```

Both suck idk why (except 3 vars...).

```
par(mfrow=c(1,3))
plot(fit.gam, se=T)
```