Assignment 4, Methods 3, 2021, autumn semester

Rikke Uldbæk

December 8th, 2021

Exercises and objectives

1) Use principal component analysis to improve the classification of subjective experience

2) Use logistic regression with cross-validation to find the optimal number of principal components

REMEMBER: In your report, make sure to include code that can reproduce the answers requested in the exercises below (**MAKE A KNITTED VERSION**)
REMEMBER: This is Assignment 4 and will be part of your final portfolio

```
#packages
pacman::p_load(reticulate, Rcpp)

#set wd
setwd("~/Desktop/Cognitive Science/3rd semester/Methods 3/github_methods_3/week_10")
```

EXERCISE 1 - Use principal component analysis to improve the classification of subjective experience

We will use the same files as we did in Assignment 3 The files `megmag_data.npy` and `pas_vector.npy` can be downloaded here (http://laumollerandersen.org/data_methods_3/megmag_data.npy) and here (http://laumollerandersen.org/data_methods_3/pas_vector.npy)
The function `equalize_targets` is supplied - this time, we will only work with an equalized data set. One motivation for this is that we have a well-defined chance level that we can compare against. Furthermore, we will look at a single time point to decrease the dimensionality of the problem

1) Create a covariance matrix, find the eigenvectors and the eigenvalues

```
#import packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.decomposition import PCA
```

```
#load data
data = np.load("/Users/rikkeuldbaek/Desktop/Cognitive Science/3rd semester/Methods 3/github_methods_3/w

y = np.load("/Users/rikkeuldbaek/Desktop/Cognitive Science/3rd semester/Methods 3/github_methods_3/week
```

i. Load `megmag_data.npy` and call it `data` using `np.load`. You can use `join`, which can be imported from `os.path`, to create paths from different string segments

```
#defined function to equalize number of targets in y and data
def equalize_targets(data, y):
    np.random.seed(7)
    targets = np.unique(y)
    counts = list()
    indices = list()
    for target in targets:
        counts.append(np.sum(y == target))
        indices.append(np.where(y == target)[0])
    min_count = np.min(counts)
    first_choice = np.random.choice(indices[0], size=min_count, replace=False)
    second_choice = np.random.choice(indices[1], size=min_count, replace=False)
    third_choice = np.random.choice(indices[2], size=min_count, replace=False)
    fourth_choice = np.random.choice(indices[3], size=min_count, replace=False)

    new_indices = np.concatenate((first_choice, second_choice,
                                  third_choice, fourth_choice))
    new_y = y[new_indices]
    new_data = data[new_indices, :, :]

    return new_data, new_y

#equalizing (overwriting) the data, so I have 98 elements of each PAS rating
data_equal, y = equalize_targets(data, y)
data_equal.shape #98 elements of each PAS rating in depth
```

ii. Equalize the number of targets in `y` and `data` using `equalize_targets`

```
## (396, 102, 251)
```

```
y.shape #98 elements of each PAS rating in depth
```

```
## (396,)
```

```
# Construct `times=np.arange(-200, 804, 4)`
#(make a vector from -200 to 800, with a value each 4th millisecond)
times = np.arange(-200, 804, 4)
```

```
#the index corresponding to 248 ms
np.where(times == 248) # index 112

# reduce the dimensionality of `data` from three to two dimensions
# by only choosing the time index corresponding to 248 ms
```

iii. Construct `times=np.arange(-200, 804, 4)` and find the index corresponding to 248 ms - then reduce the dimensionality of `data` from three to two dimensions by only choosing the time index corresponding to 248 ms (248 ms was where we found the maximal average response in Assignment 3)

```
## (array([112]),)
```

```
data= data_equal[:,:,112]
data.shape # two dimensions now
```

```
## (396, 102)
```

```
#standardizing the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)
scaled_data.shape
```

iv. Scale the data using `StandardScaler`

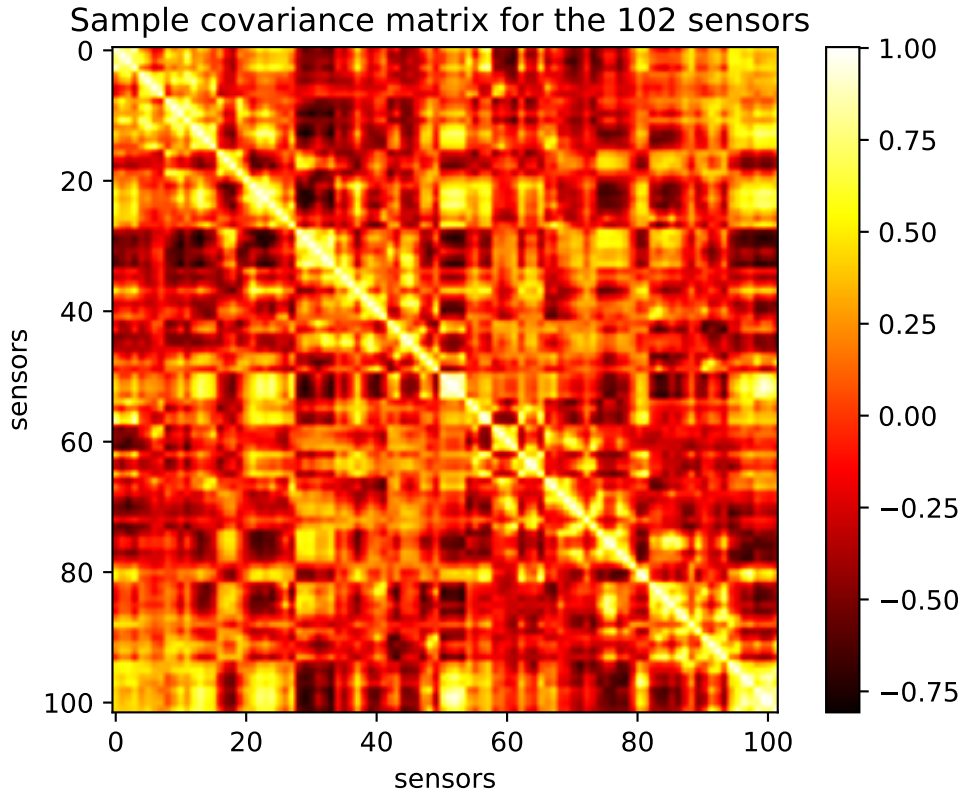
```
## (396, 102)
```

```
# Create the sensor covariance matrix
cov_mat = np.cov(scaled_data.T)

#plot the cov_mat for the sensors (102 per axis)
import matplotlib.pyplot as plt

plt.figure()
plt.imshow(cov_mat,cmap = "hot")
plt.colorbar()
```

v. Calculate the sample covariance matrix for the sensors (you can use `np.cov`) and plot it (either using `plt.imshow` or `sns.heatmap` (`import seaborn as sns`))

```
## <matplotlib.colorbar.Colorbar object at 0x166cc4ee0>
```

```
plt.title("Sample covariance matrix for the 102 sensors ")
plt.xlabel('sensors')
plt.ylabel('sensors')
plt.show()
```



vi. What does the off-diagonal activation imply about the independence of the signals measured by the 102 sensors? The off-diagonal activation imply that the 102 sensors do not really pick up independent signals, meaning that some of the sensors probably pick up some of the same signals. This is made clear by the plot of the covariance matrix, where we see a lot of values in the off-diagonal that are non-zero (there are both high positive and negative covariance, as specified by the colorbar).

vii. Run `np.linalg.matrix_rank` on the covariance matrix - what integer value do you get? (we'll use this later) When using the `np.linalg.matrix_rank` on my covariance matrix I get the integer value 97.

```
#Run `np.linalg.matrix_rank` on the covariance matrix
int_val = np.linalg.matrix_rank(cov_mat)
print("The integer value is", int_val)
```

```
## The integer value is 97
```

```
#Find the eigenvalues and eigenvectors of the covariance matrix using `np.linalg.eig`
from numpy.linalg import eig as eigenValuesAndVectors

eigen_values, eigen_vectors = eigenValuesAndVectors(cov_mat)

# Due to complexity of the numbers, use `np.real` to retrieve only the real parts
eigen_values = np.real(eigen_values)
eigen_vectors = np.real(eigen_vectors)
```

viii. Find the eigenvalues and eigenvectors of the covariance matrix using `np.linalg.eig` - note that some of the numbers returned are complex numbers, consisting of a real and an imaginary part (they have a j next to them). We are going to ignore this by only looking at the real parts of the eigenvectors and -values. Use `np.real` to retrieve only the real parts

2) Create the weighting matrix W and the projected data, Z

i. We need to sort the eigenvectors and eigenvalues according to the absolute values of the eigenvalues (use `np.abs` on the eigenvalues).

```
#use `np.abs` on the eigenvalues
eigen_values= np.abs(eigen_values)
```

```
#find the correct ordering of the indices and create
# an array, e.g. `sorted_indices` that contains these indices.
sorted_indices= np.argsort(eigen_values)
sorted_indices = np.flip(sorted_indices) #lowest to highest
sorted_indices
```

ii. Then, we will find the correct ordering of the indices and create an array, e.g. `sorted_indices` that contains these indices. We want to sort the values from highest to lowest. For that, use `np.argsort`, which will find the indices that correspond to sorting the values from lowest to highest. Subsequently, use `np.flip`, which will reverse the order of the indices.

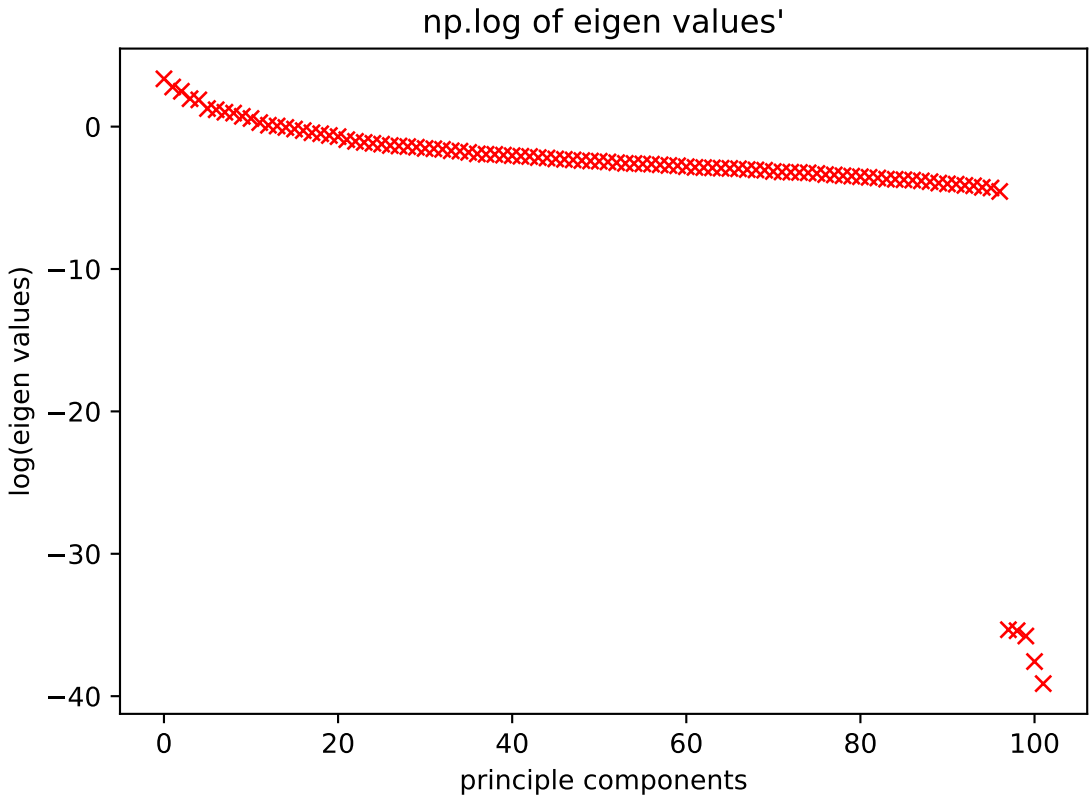
```
## array([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,
##         13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,
##         26,  27,  28,  29,  31,  30,  32,  34,  33,  35,  36,  37,  38,
##         39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
##         52,  53,  54,  56,  55,  57,  58,  59,  61,  64,  66,  71,  72,
##         73,  74,  76,  77,  78,  82,  93,  91,  94,  87,  86,  95,  90,
##         84,  85,  92,  88,  83,  96,  89,  81,  80,  79,  75,  70,  69,
##         68,  67,  65,  63,  62,  60, 101,  97, 100,  99,  98])
```

```
#create arrays of sorted eigenvalues and eigenvectors
# using the `sorted_indices` array just created
eigen_values = eigen_values[sorted_indices]
eigen_vectors = eigen_vectors[:,sorted_indices]
```

iii. Finally, create arrays of sorted eigenvalues and eigenvectors using the `sorted_indices` array just created. For the eigenvalues, it should like this `eigenvalues = eigenvalues[sorted_indices]` and for the eigenvectors: `eigenvectors = eigenvectors[:, sorted_indices]`

iv. Plot the log, `np.log`, of the eigenvalues, `plt.plot(np.log(eigenvalues), 'o')` - are there some values that stand out from the rest? In fact, 5 (noise) dimensions have already been projected out of the data - how does that relate to the matrix rank (Exercise 1.1.vii) Yes 5 values stand out from the rest. When looking at the sorted_indices it is evident that the last 5 values are much more negative (-35 to -39) than the rest 97 values. In fact, these 97 values are directly related to the 97 values in the matrix rank.

```
#plot the log of the eigenvalues
plt.figure()
plt.plot(np.log(eigen_values), 'rx')
plt.title("np.log of eigen values'")
plt.ylabel("log(eigen values)")
plt.xlabel("principle components")
plt.show()

#logged version for interpretation
```




```
log_eigen = np.log(eigen_values)
log_eigen[sorted_indices] #the five last values are extremely negative
```

```
## array([  3.35674527,   2.78212014,   2.48213471,   1.96283254,
```

```
##          1.87756779,   1.27117459,   1.1839596 ,   1.01048866,
##          0.948959  ,   0.72238718,   0.5700768 ,   0.28292523,
##          0.09604088,   0.04305278,  -0.05864892,  -0.16988542,
##         -0.27964178,  -0.42950715,  -0.50004516,  -0.62360666,
##         -0.70384654,  -0.91615604,  -1.03417312,  -1.10980711,
##         -1.17297602,  -1.23151451,  -1.32730589,  -1.36219089,
##         -1.40232561,  -1.44736119,  -1.55341719,  -1.53346189,
##         -1.59666964,  -1.72256598,  -1.69768001,  -1.80960706,
##         -1.91363889,  -1.94362535,  -1.95069178,  -2.00087052,
##         -2.03933315,  -2.08101912,  -2.0976541 ,  -2.17395025,
##         -2.20176023,  -2.26938203,  -2.30012474,  -2.33705031,
##         -2.38338011,  -2.41725376,  -2.43001836,  -2.4791752 ,
##         -2.53186376,  -2.58136232,  -2.59020077,  -2.63452201,
##         -2.61808854,  -2.68200561,  -2.72877278,  -2.74791774,
##         -2.86253059,  -2.92204887,  -2.96058587,  -3.18353441,
##         -3.20050381,  -3.21906068,  -3.24414889,  -3.3687465 ,
##         -3.39192413,  -3.4491636 ,  -3.6061307 ,  -4.15541837,
##         -4.04375743,  -4.26527994,  -3.8107665 ,  -3.75725969,
##         -4.31097805,  -4.01937377,  -3.70041514,  -3.73026859,
##         -4.12822397,  -3.86711463,  -3.68987353,  -4.55390497,
##         -3.96450858,  -3.56092334,  -3.52113974,  -3.47769164,
##         -3.28946054,  -3.15183408,  -3.06290895,  -3.04080583,
##         -3.0158261 ,  -2.93611107,  -2.89668977,  -2.87602846,
##         -2.81523991, -39.11907887, -35.33275964, -37.5691422 ,
##        -35.77607067, -35.40311244])
```

```
#Create the weighting matrix, `W`
W = eigen_vectors[:,sorted_indices]
```

v. Create the weighting matrix, `W` (it is the sorted eigenvectors)

```
#Create the projected data, `Z`
Z  = scaled_data @ W #where scaled data is my X

#check But bro does not make sense
X_check = Z @ W.T #how to make X
np.isclose(X_check, scaled_data)  # needs to be TRUE,
```

vi. Create the projected data, `Z`, $Z = XW$ - (you can check you did everything right by checking whether the X you get from $X = ZW^T$ is equal to your original X, `np.isclose` may be of help)'

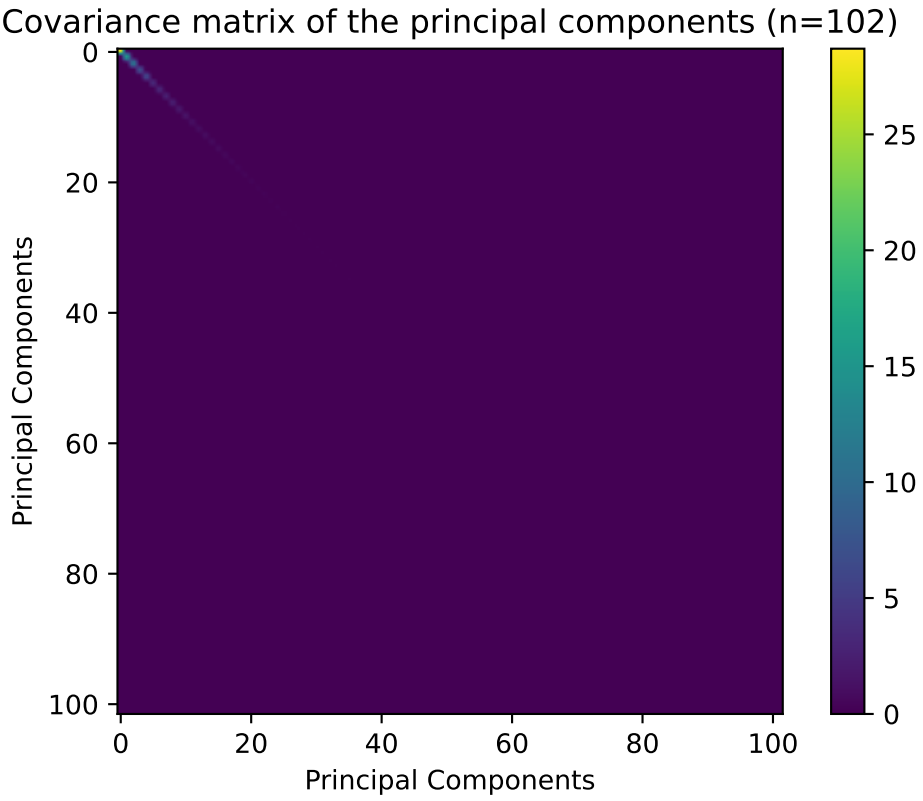
```
## array([[ True,  True,  True, ...,  True,  True,  True],
##        [ True,  True,  True, ...,  True,  True,  True],
##        [ True,  True,  True, ...,  True,  True,  True],
##        ...,
##        [ True,  True,  True, ...,  True,  True,  True],
##        [ True,  True,  True, ...,  True,  True,  True],
##        [ True,  True,  True, ...,  True,  True,  True]])
```

vii. Create a new covariance matrix of the principal components (n=102) - plot it! What has happened off-diagonal and why? After we've performed the PCA on the covariance matrix, we can see that almost all of the covariance from the off-diagonal has been removed - this is due to the nature of Principal component analysis, which bundles together predictor variables that explains a lot of the same variance into one new principal component, that is made to minimize the covariance with the other principal components. The reason why we don't see any diagonal activation, is due to the very small numbers, it's only the first few principle components that we can see the covariance of, because the principle components are ordered in the rank of how much variance they explain.

```
# Create the eigen covariance matrix
eigen_cov_mat = np.cov(Z.T)

#plot the eigen cov_mat of the principal components (n=102)
import matplotlib.pyplot as plt

plt.figure()
plt.imshow(eigen_cov_mat)
plt.colorbar()
```

```
## <matplotlib.colorbar.Colorbar object at 0x166ec9c10>
```

```
plt.title("Covariance matrix of the principal components (n=102)")
plt.xlabel('Principal Components')
plt.ylabel('Principal Components')
plt.show()

#as we see here, some of the first priciple components are huge compared to the last ones
```




```
eigen_cov_mat[1,1] #16.153231721724463 (diagonal activation), large
```

```
## 16.153231721724463
```

```
eigen_cov_mat[90,90] #0.04675149941476335 (diagonal activation), small
```

```
## 0.04675149941476335
```

```
eigen_cov_mat[90,70] # 1.5725817272855065e-16 (off-diagonal activation), even smaller!
```

```
## 1.5725817272855065e-16
```

EXERCISE 2 - Use logistic regression with cross-validation to find the optimal number of principal components

1) We are going to run logistic regression with in-sample validation

```
## run logistic regression with in-sample validation
scores_1 = []
logReg = LogisticRegression(penalty ='none', solver= 'newton-cg', random_state=1)
```

```
#loop through 102 models NORMAL SCORE
for i in range(102):
  logReg.fit(Z[:,0:i+1], y) #
  fit_score = logReg.score(Z[:,0:i+1], y)
  scores_1.append(fit_score)
```

```
# minimum and maximum values
np.min(scores_1)
```

i. First, run standard logistic regression (no regularization) based on $Z_{d\times k}$ and y (the target vector). Fit (`.fit`) 102 models based on: $k = [1, 2, ..., 101, 102]$ and $d = 102$. For each fit get the classification accuracy, (`.score`), when applied to $Z_{d\times k}$ and y. This is an in-sample validation. Use the solver `newton-cg` if the default solver doesn't converge'

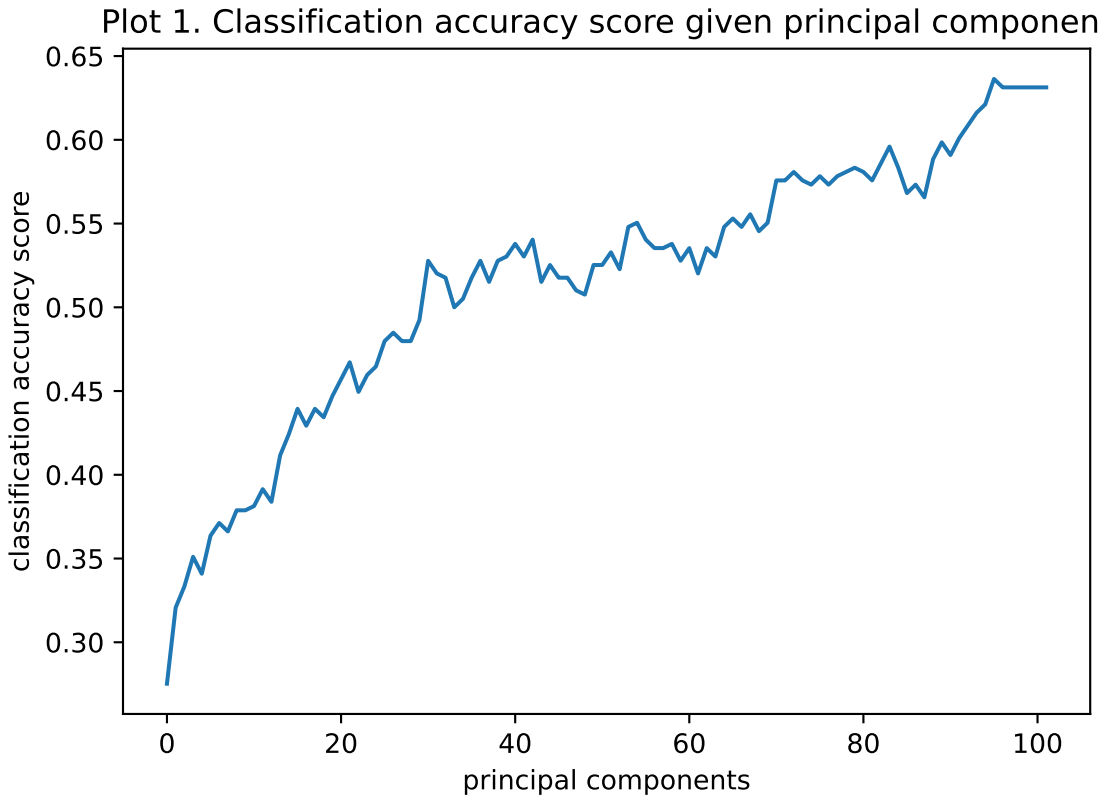
```
## 0.27525252525252525
```

```
np.max(scores_1)
```

```
## 0.6363636363636364
```

```
#plot 1 (only logistic regression), x=  principal components (102), y= classification accuracy
plt.figure()
plt.plot(scores_1)
plt.xlabel("principal components")
plt.ylabel("classification accuracy score")
plt.title("Plot 1. Classification accuracy score given principal component")
plt.show()
```

ii. Make a plot with the number of principal components on the x-axis and classification accuracy on the y-axis - what is the general trend and why is this so?



iii. In terms of classification accuracy, what is the effect of adding the five last components? Why do you think this is so? Adding the five last components have very little effect on the classification accuracy as visualized in the plot. The last 5 principle components explain very little variance, if any at all. Therefore it makes sense, when adding them in the logistic regression, that they do not contribute to the classification accuracy.

2) Now, we are going to use cross-validation - we are using `cross_val_score` and `StratifiedKFold` from `sklearn.model_selection`

```
# Define the variable: `cv = StratifiedKFold()`
cv = StratifiedKFold(n_splits=5)

## use cross-validation this time
cv_scores = []
logReg = LogisticRegression(penalty ='none', solver= 'newton-cg', random_state=1)

#loop through 102 models NORMAL SCORE
for i in range(102):
  logReg.fit(Z[:,0:i+1], y)
  cross_val_scores = cross_val_score(logReg, Z[:,0:i+1], y, cv=cv) #cv being 5
  cv_scores.append(np.mean(cross_val_scores))
```

```
# minimum and maximum values
np.min(cv_scores)
```

i. Define the variable: `cv = StratifiedKFold()` and run `cross_val_score` (remember to set the `cv` argument to your created `cv` variable). Use the same `estimator` in `cross_val_score` as in Exercise 2.1.i. Find the mean score over the 5 folds (the default of `StratifiedKFold`) for each k, $k = [1, 2, ..., 101, 102]$

```
## 0.23474683544303798
```

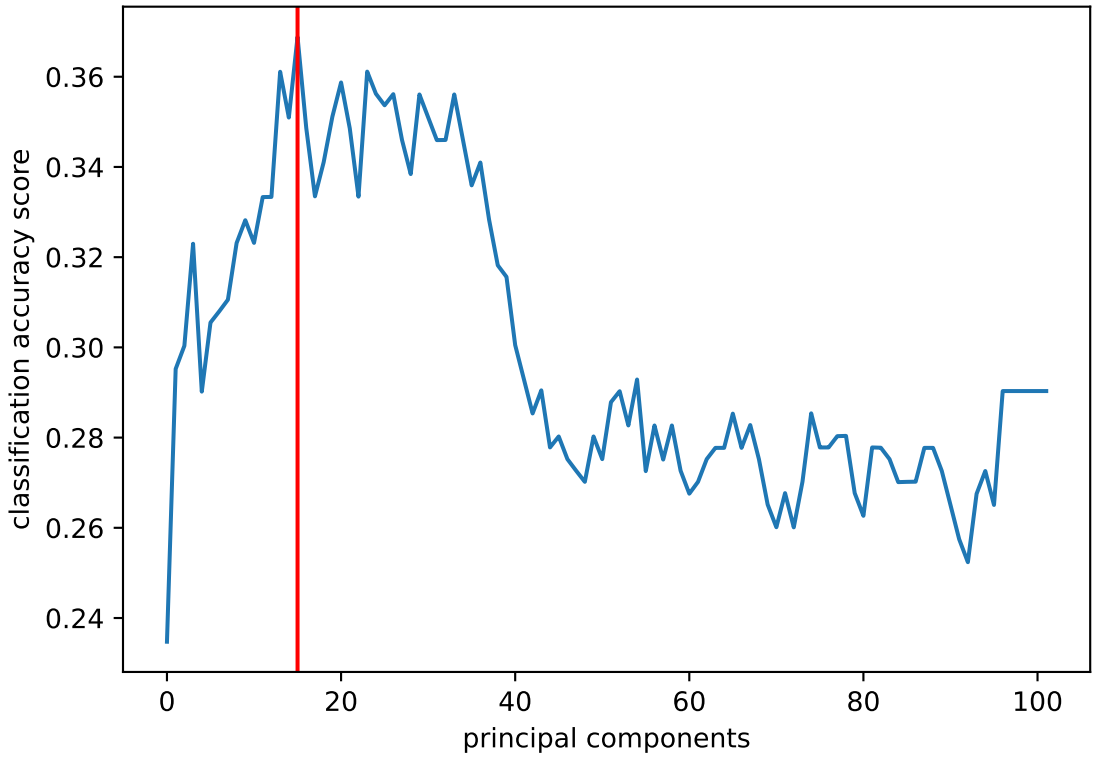
```
np.max(cv_scores)
```

```
## 0.3688291139240506
```

ii. Make a plot with the number of principal components on the x-axis and classification accuracy on the y-axis - how is this plot different from the one in Exercise 2.1.ii? In plot 1 it looks like the classification accuracy score gradually becomes better when adding a principle component at each fit. Using cross validation when fitting the logistic regression, plot 2 shows a more accurate distribution of the classification accuracy score for the principle components, since we cross validate with 5 folds, and therefore let the model train and test 5 times on different parts of the data. This way we get a better indication of how well our model will perform on data it has not seen before, unlike running logistic regression with in-sample validation. Note, the scale of classification score on the y-axis is very different from plot 1 and plot 2, in plot 1 it ranges from around 0.27 to 0.63, while in plot 2 it ranges from 0.23 to 0.37.

```
#plot 2 (using cross validation), x=  principal components (102), y= classification accuracy
plt.figure()
plt.plot(cv_scores)
plt.axvline(np.argmax(cv_scores), color= "red")
plt.xlabel("principal components")
plt.ylabel("classification accuracy score")
plt.title("Plot 2. Classification accuracy score given principal component, using CV")
plt.show()
```

Plot 2. Classification accuracy score given principal component, using CV



iii. What is the number of principal components, $k_{max\_accuracy}$, that results in the greatest classification accuracy when cross-validated? The first 16 principle components (indexed from 0) result in the greatest classification accuracy when cross-validated.

```
#the maximal cross validation score the model
np.amax(cv_scores) #0.3688291139240506
```

```
## 0.3688291139240506
```

```
np.argmax(cv_scores)  # PC 15
```

```
## 15
```

iv. How many percentage points is the classification accuracy increased with relative to the to the full-dimensional, d, dataset The increase of classification accuracy in percentage points is 7%.

```
# Runing cross validation model on full dimensional data
full_data_cv_scores = cross_val_score(logReg, scaled_data, y, cv=cv)
np.mean(full_data_cv_scores) #0.2903164556962025

#the maximal cross validation score the model (for comparison)
```

```
## 0.2903164556962025
```

```
np.amax(cv_scores) #0.3688291139240506

#increase of classification accuracy in percentage points
```

```
## 0.3688291139240506
```

```
np.amax(cv_scores) -np.mean(full_data_cv_scores)  #0.07851265822784809
```

```
## 0.07851265822784809
```

v. How do the analyses in Exercises 2.1 and 2.2 differ from one another? Make sure to comment on the differences in optimization criteria. In exercise 2.1 we use logistic regression with in-sample validation, which means our train and test data is unchanged throughout the modelling, i.e. we use the same training data to predict the the same test data throughout the entire analysis. This means the model only becomes really good at predicting that specific test set (as visualized in plot 1) but will perform worse when predicting unseen new data.

In exercise 2.2 we use cross-validation with 5 folds, which means that we split our whole data into training and test set through 5 iterations (5 folds) and get a performance measure (classification accuracy score) for each fold. Then we take the mean of the performance measure from the 5 folds in order to evaluate our model. This way the model becomes better at predicting unseen new data. What constitutes a better model is the classification accuracy of new unseen data, therefore cross validation is the preferred analysis in this case.

What constitutes a better model is the classification accuracy of new unseen data, therefore cross validation is the preferred analysis in this case, where with the other method, you end up with overfitting.

3) We now make the assumption that $k_{max\_accuracy}$ is representative for each time sample (we only tested for 248 ms). We will use the PCA implementation from *scikit-learn*, i.e. import `PCA` from `sklearn.decomposition`.

```
#my model from before
logReg = LogisticRegression(penalty ='none', solver= 'newton-cg', random_state=1)



############## analysis 1 - reduced dimensions = 16
pca_15 = PCA(n_components=15) #number of components
cv_scores_pca_15 =  [] #make an empty list

for i in range(251):
  transformed_reduced_data = scaler.fit_transform(data_equal[:,:,i])
  # transform the data first
  PCA_15 = pca_15.fit_transform(transformed_reduced_data)
  logReg.fit(PCA_15, y)
  #loop and fit through the transformed data
  pca_scores_1= cross_val_score(logReg, PCA_15, y, cv=5)
  #calculate cross validation scores
  cv_scores_pca_15.append(np.mean(pca_scores_1))
```

```
  #append to the list 'cv_scores'


############ analysis 2 - All data
```

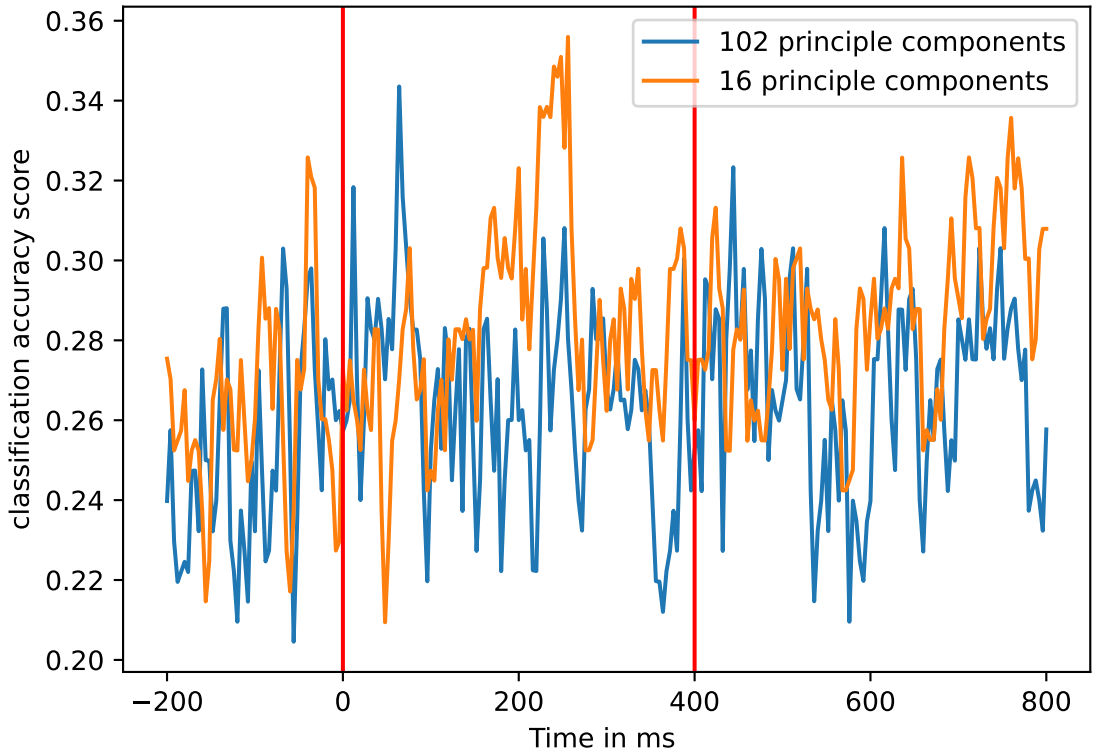
```
pca_102 = PCA(n_components=102) #number of components
cv_score_pca_102 = [] #make an empty list

for i in range(251):
  transformed_all_data = scaler.fit_transform(data_equal[:,:,i])
  # transform the data first
  PCA_102 = pca_102.fit_transform(transformed_all_data)
  logReg.fit(PCA_102, y)
  #loop and fit through the transformed data
  pca_scores_2= cross_val_score(logReg, PCA_102, y, cv=5)
  #calculate cross validation scores
  cv_score_pca_102.append(np.mean(pca_scores_2))
  #append to the list 'cv_scores'
```

i. For each of the 251 time samples, use the same estimator and cross-validation as in Exercises 2.1.i and 2.2.i. Run two analyses - one where you reduce the dimensionality to $k_{max\_accuracy}$ dimensions using `PCA` and one where you use the full data. Remember to scale the data (for now, ignore if you get some convergence warnings - you can try to increase the number of iterations, but this is not obligatory)

```
#plot 3 (using cross validation), x=  principal components (102), y= classification accuracy
plt.figure()
plt.plot(times,cv_score_pca_102)
plt.plot(times,cv_scores_pca_15)
plt.xlabel("Time in ms ")
plt.ylabel("classification accuracy score")
plt.title("Plot 3. Classification accuracies of the analysis with 102 and 16 components ")
plt.legend(["102 principle components", "16 principle components"])
plt.axvline((times[150]), color= "red")
plt.axvline((times[50]), color= "red")
plt.show()

#maximum classification scores
```

ii. Plot the classification accuracies for each time sample for the analysis with PCA and for the one without in the same plot. Have time (ms) on the x-axis and classification accuracy on the y-

Plot 3. Classification accuracies of the analysis with 102 and 16 components



axis

```
np.max(cv_scores_pca_15) #0.3559493670886076
```

```
## 0.3559493670886076
```

```
np.max(cv_score_pca_102) #0.3435443037974684
```

```
## 0.3435443037974684
```

iii. Describe the differences between the two analyses - focus on the time interval between 0 ms and 400 ms - describe in your own words why the logistic regression performs better on the PCA-reduced dataset around the peak magnetic activity The main difference between the two analysis, when focusing on the time interval 0ms - 400 ms, is the number of principal components used when classifying. According to the plot 3, it seems like the first 16 principle components (indexed from 0) result in the greatest classification accuracy of 36%, while the total number of principle components (102) performs only a little worse with a maximal classification accuracy of 34%. Even though the 16 principle components seem to perform better, it is in fact only a slightly better performance than the 102 principle components.

The model performs better on the PCA-reduced dataset (16 principle components), because we got rid of covariated variables which did not contribute in the classification. This subset of PCA-reduced data does not contain any disturbing noise from the remaining principle components, and only contains what actually contributes most to the classification.