Portfolio 4, Methods 3, 2021, autumn semester

LAURA W. PAABY

1/12 - 2O21

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

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, wewill look at a single time point to decrease the dimensionality of the problem

In [ ]:

```
###LOADING LIBRARIES
## importing libraries
import statistics
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import RANSACRegressor
from sklearn.model_selection import train_test_split
import scipy as sp
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.linear_model import ElasticNet
from sklearn.preprocessing import PolynomialFeatures
from sklearn.ensemble import RandomForestRegressor
import os
```

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

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

In [ ]:

```
# IMPORTING DATA
import requests
import io

# THE MEG DATA
response = requests.get('http://laumollerandersen.org/data_methods_3/megmag_d
response.raise_for_status()
data = np.load(io.BytesIO(response.content))

#numbers of repitions of visual stimuli: 682
#numbers of sensors (spots in the cap that record magnetic fields): 102
#numbers of time samples: 251



# THE PAS VECTOR
response = requests.get('http://laumollerandersen.org/data_methods_3/pas_vect
response.raise_for_status()
y = np.load(io.BytesIO(response.content))
```

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

In [ ]:

```
### DEFINING THE EQUIALIZER FUNCTION
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=Fals
    third_choice = np.random.choice(indices[2], size=min_count, replace=False
    fourth_choice = np.random.choice(indices[3], size=min_count, replace=Fals
    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
```

In [ ]:

```
## EQUALIZING THE DATA AND Y:
```

```
data_equal, y_equal = equalize_targets(data, y)
```

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)

In [ ]:
```
#making the time array
times=np.arange(-200, 804, 4)
```

In [ ]:
```
## finding the time
time248 = np.argwhere(times == 248)
print("the time index corresponding to 248 ms:",time248)
```

```
the time index corresponding to 248 ms: [[112]]
```

In [ ]:
```
## squeezing the data to two dim with the time point in the data:
data_time = data_equal[:,:,time248].squeeze()
print("the data is now two dimensional:", data_time.shape)
```

```
the data is now two dimensional: (396, 102)
```

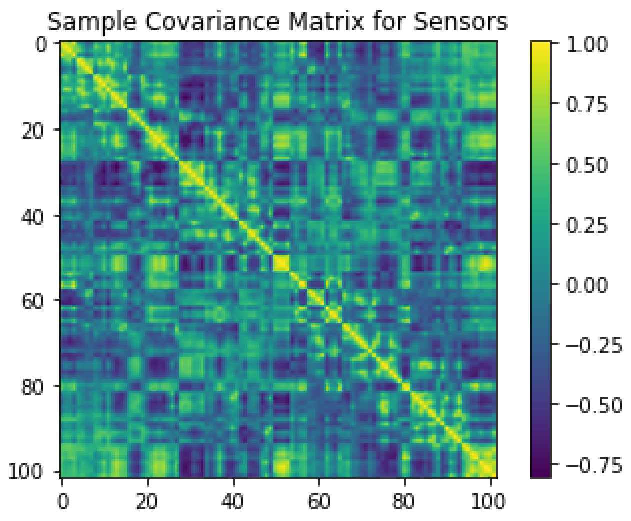
iv. Scale the data using StandardScaler

In [ ]:
```
# standardizing
sc = StandardScaler()
data_time_std = sc.fit_transform(data_time)
```

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))

In [ ]:
```
cov_mat = np.cov(data_time_std.T)
plt.imshow(cov_mat)
plt.title("Sample Covariance Matrix for Sensors")
plt.colorbar()

#the shape of features
print(cov_mat.shape)
```

```
(102, 102)
```



vi. What does the off-diagonal activation imply about the independence of the signals measured by the 102 sensors?

There appears to be a rather high covariance in the off-diagonal activation, looking at the colors. We see how many matches the colours on the scale around -0.75 and 1, indicating high covariance. The independece signals are therefore quite low, we'd argued

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

In [ ]:
```python
np_lin_matrix = np.linalg.matrix_rank(cov_mat)
print("the rank of the cov matrix:", np_lin_matrix)
```

```
the rank of the cov matrix: 97
```

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

In [ ]:
```python
### finding eigen values and vectors
eigen_values_mat, eigen_vectors_mat = np.linalg.eig(cov_mat)
print(eigen_values_mat, eigen_vectors_mat)

eigen_values_mat = np.real(eigen_values_mat)
eigen_vectors_mat = np.real(eigen_vectors_mat)
```

```
[ 2.86956421e+01  1.61532317e+01  1.19667828e+01  7.11946468e+00
  6.53758473e+00  3.56503758e+00  3.26728576e+00  2.74694300e+00
  2.58301935e+00  2.05934337e+00  1.76840286e+00  1.32700594e+00
  1.10080407e+00  1.04399299e+00  9.43037791e-01  8.43761487e-01
  7.56054528e-01  6.50829775e-01  6.06503267e-01  5.36007750e-01
  4.94678836e-01  4.00053880e-01  3.55520231e-01  3.29622537e-01
  3.09444655e-01  2.91850233e-01  2.65190752e-01  2.56099076e-01
  2.46024142e-01  2.35190093e-01  2.11523920e-01  2.15787340e-01
  2.02570026e-01  1.78607257e-01  1.83107840e-01  1.63718456e-01
  1.47542517e-01  1.43183916e-01  1.42175684e-01  1.35217523e-01
  1.30115449e-01  1.24802958e-01  1.22744036e-01  1.13727476e-01
  1.10608291e-01  1.03376043e-01  1.00246339e-01  9.66121945e-02
  9.22382747e-02  8.91661528e-02  8.80352165e-02  8.38123252e-02
  7.95106934e-02  7.56708457e-02  7.50049795e-02  7.17532589e-02
  7.29421553e-02  6.84257809e-02  6.52993774e-02  6.40611140e-02
  1.05260202e-02  5.98903487e-02  1.34204174e-02  1.40479340e-02
  5.71240196e-02  1.56792301e-02  5.63581476e-02  1.61114679e-02
  1.75314752e-02  1.79642111e-02  1.89773602e-02  5.52056614e-02
  5.38232974e-02  5.30717201e-02  5.17885667e-02  2.09186404e-02
  4.90053356e-02  4.77963583e-02  4.67514994e-02  2.21312089e-02
  2.33476326e-02  2.39863924e-02  4.27736047e-02  2.71567213e-02
  3.17721994e-02  3.08786078e-02  3.72739518e-02  3.90017448e-02
  2.84125782e-02  2.47132649e-02  3.36438792e-02  4.07416727e-02
  2.95657187e-02  4.14389338e-02  3.99926065e-02  3.44327716e-02
  2.49751605e-02  4.21331203e-16  1.02517949e-17 -4.82979671e-17
 -2.90168273e-16 -4.52040614e-16] [[ 0.12771427 -0.06908204 -0.08059849 ... -
0.05359027 -0.18106345
   0.0759772 ]
 [ 0.09676535 -0.14564561 -0.07476254 ... -0.02035155 -0.10215784
   0.02548895]
 [ 0.10334695 -0.12256363 -0.12387986 ... -0.02430483 -0.12870615
   0.04402212]
 ...
 [ 0.16293631  0.04934011  0.06496482 ...  0.03952017  0.15032851
   0.15720407]
 [ 0.15957395  0.03865037  0.04133159 ...  0.13440067  0.16404444
   0.08148025]
 [ 0.14719124 -0.03554908  0.0728808  ...  0.0327546   0.08479141
   0.11853622]]
```

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).

In [ ]:
```
abs_eigen_val = abs(eigen_values_mat)
```

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.

In [ ]:
```
sorted_indices = np.flip(np.argsort(abs_eigen_val))
print(sorted_indices)
```

```
[  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]
```

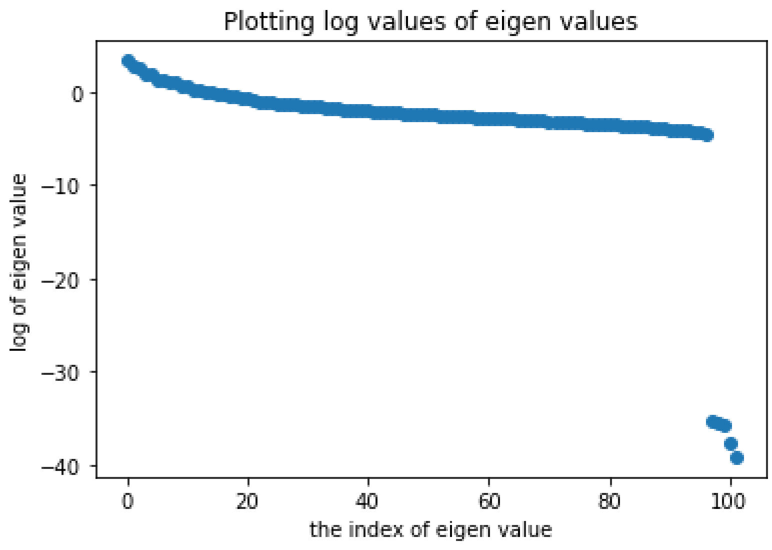
iii. Finally, create arrays of sorted eigenvalues and eigenvectors using the sorted_indices array just created. For the eigenvalues, it should like this eigenvalues = eigenvalues[sorted_indices] and for the eigenvectors: eigenvectors = eigenvectors[:, sorted_indices]

In [ ]:
```
eigen_val_sorted = abs_eigen_val[sorted_indices]
eigen_vec_sorted = eigen_vectors_mat[:,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)

In [ ]:
```
log_eigen = np.log(eigen_val_sorted)
plt.plot((log_eigen), 'o')
plt.title("Plotting log values of eigen values")
plt.xlabel("the index of eigen value")
plt.ylabel("log of eigen value")
```

Out[ ]:
```
Text(0, 0.5, 'log of eigen value')
```



There appears to be some values around index 100, where the last 5 values stands out (they are below -30 on the log scale, while the rest is above -10), maybe because the variance information these contains are already explained by the 97 other dimensions. The matrix rank it self was 97, meaning that 97 columns are linearly independent, and can explain all the variance, leaving the 5 (from 97 to 102) dimensions with a very low log value.

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

In [ ]:
```
W = eigen_vec_sorted
```

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)

In [ ]:
```
X = data_time_std #saving the data in matrix X

#creating the projected data z:
Z = X @ W
```

In [ ]:
```
## checking if its equal to the original:
check_X = Z @ W.T
np.isclose(check_X, X)
```

Out[ ]:
```
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?

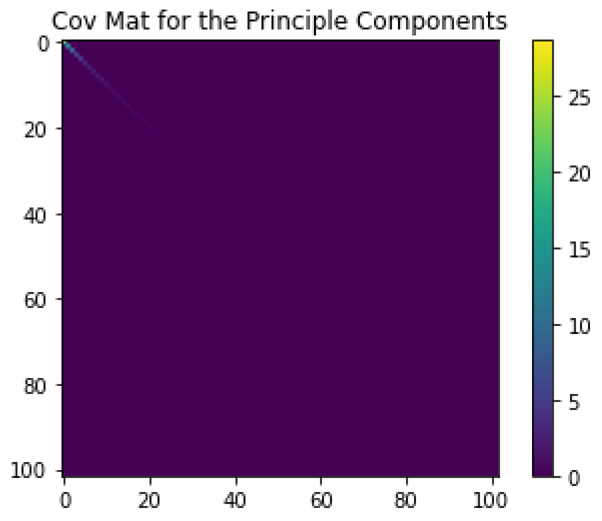
In [ ]:
```
print(W.shape)
```

```
(102, 102)
```

In [ ]:
```
cov_mat_pc = np.cov(Z.T)
```

```
plt.imshow(cov_mat_pc)
plt.colorbar()
plt.title("Cov Mat for the Principle Components")

#the shape of features
print(cov_mat_pc.shape)
```

```
(102, 102)
```



There does not seem to be any covariance overall by inspecting the plot, which makes sense since the principle components are orthogonal. However, the few first principle components seem to covary with themselves, which could be explained by the squared eigenvalues that express variance.

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

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

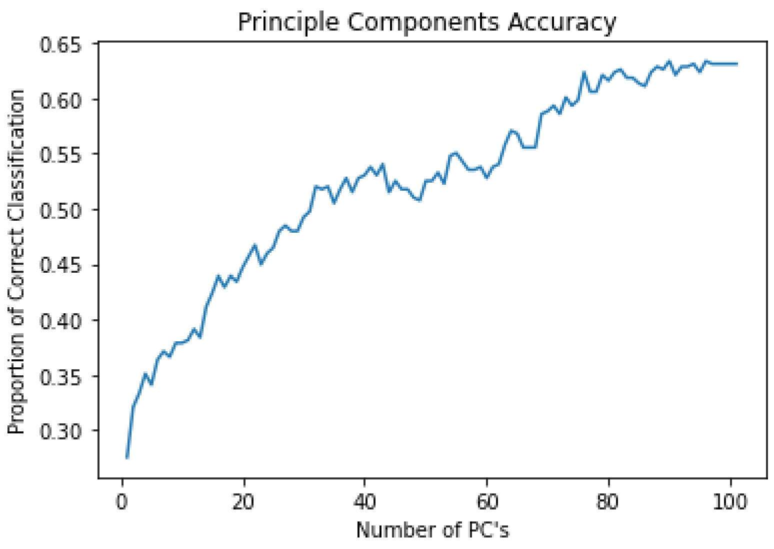
In [ ]:
```
k_score = []
for i in range(101):
    log_r = LogisticRegression(penalty= 'none', solver= "newton-cg")
    log_r.fit(Z[:,0:i+1], y_equal)
    score = log_r.score(Z[:,0:i+1], y_equal)
    k_score.append(score)
```

In [ ]:
```
k_score = np.array(k_score)
print("the classification accuracy scores:", k_score)
print(k_score.shape)
```

```
the classification accuracy scores: [0.27525253 0.32070707 0.33333333 0.351010
1  0.34090909 0.36363636
 0.37121212 0.36616162 0.37878788 0.37878788 0.38131313 0.39141414
 0.38383838 0.41161616 0.42424242 0.43939394 0.42929293 0.43939394
 0.43434343 0.4469697  0.45707071 0.46717172 0.44949495 0.45959596
 0.46464646 0.47979798 0.48484848 0.47979798 0.47979798 0.49242424
 0.49747475 0.52020202 0.51767677 0.52020202 0.50505051 0.51767677
 0.52777778 0.51515152 0.52777778 0.53030303 0.53787879 0.53030303
 0.54040404 0.51515152 0.52525253 0.51767677 0.51767677 0.51010101
 0.50757576 0.52525253 0.52525253 0.53282828 0.52272727 0.5479798
 0.55050505 0.54292929 0.53535354 0.53535354 0.53787879 0.52777778
 0.53787879 0.54040404 0.55808081 0.57070707 0.56818182 0.55555556
 0.55555556 0.55555556 0.58585859 0.58838384 0.59343434 0.58585859
 0.6010101  0.59343434 0.59848485 0.62373737 0.60606061 0.60606061
 0.62121212 0.61616162 0.62373737 0.62626263 0.61868687 0.61868687
 0.61363636 0.61111111 0.62373737 0.62878788 0.62626263 0.63383838
 0.62121212 0.62878788 0.62878788 0.63131313 0.62373737 0.63383838
 0.63131313 0.63131313 0.63131313 0.63131313 0.63131313]
(101,)
```

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?

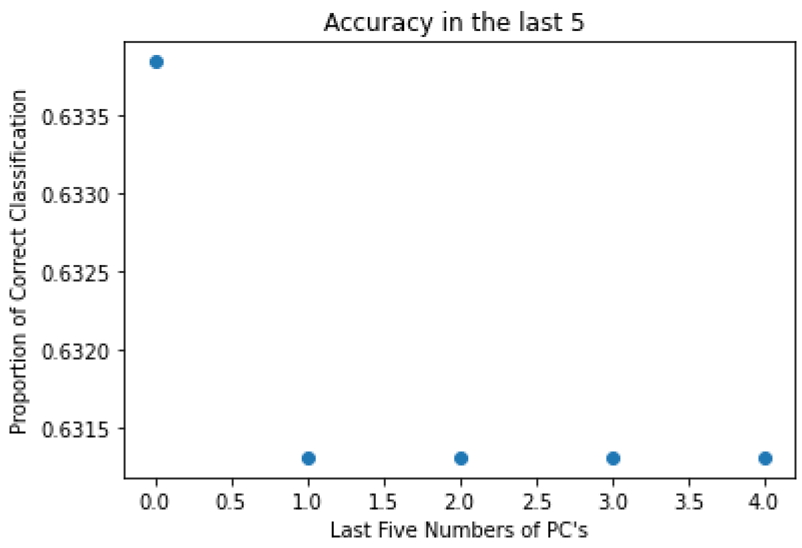
In [ ]:
```python
## remember the eigenvectors/principle components is saved in W:
#plotting
pc_list = np.arange(1, 102, 1)


plt.plot(pc_list, k_score)
plt.xlabel("Number of PC's")
plt.ylabel("Proportion of Correct Classification")
plt.title("Principle Components Accuracy ")
plt.show()
```



The trend appears to be that the accucary of the classification increases with the value of the principle components. However, adding the five last components have very little effect on the accuracy of classification. This could be due to the fact that the regression gets better to fit the components is added, which eventually leads to overfitting (since we don't cross validate). Around the 100th principle components, we see how the line flattens. This might be the case as we previously noticed how these five dimensions explain very little variance, if any at all. Adding these to the regression does therefore not making the regression better in classifying. This is visualised in the next plot;

iii. In terms of classification accuracy, what is the effect of adding the five last components? Why do you think this is so?

In [ ]:
```
plt.plot(k_score[-6:-1], 'o')
plt.xlabel("Last Five Numbers of PC's")
plt.ylabel("Proportion of Correct Classification")
plt.title("Accuracy in the last 5 ")
plt.show()
```



We see how the accuracy decreases over the last five principal components, which makes sense since these PC's not contains any variance useful to the classification.

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

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, $k = [1, 2, \dots, 101, 102]$

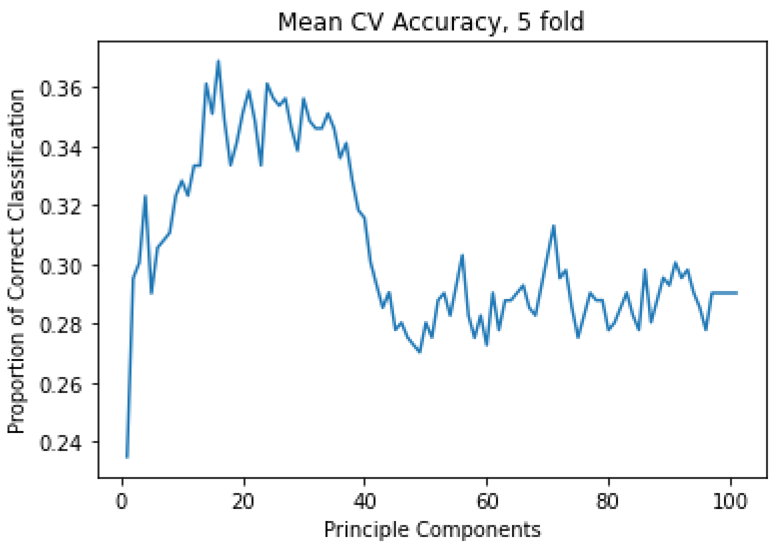
In [ ]:
```
from sklearn.model_selection import cross_val_score, StratifiedKFold
```

In [ ]:
```
#getting the cross val scores
cv_score = []
cv = StratifiedKFold()
for i in range(101):
  logR = LogisticRegression(penalty='none', solver = 'newton-cg', random_stat
  logR.fit(Z[:,0:i+1],y_equal)
  scores = cross_val_score(logR,Z[:,0:i+1],y_equal, cv = 5 )
  mean_cv_score = np.mean(scores)
  cv_score.append(mean_cv_score)

cv_score_array = np.array(cv_score)
```

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 [ ]:
```
plt.plot(pc_list, cv_score_array)
plt.title("Mean CV Accuracy, 5 fold")
plt.xlabel("Principle Components")
plt.ylabel("Proportion of Correct Classification")
```

Out[ ]:
```
Text(0, 0.5, 'Proportion of Correct Classification')
```



In the first plot it appears as if the classification accuracy score gradually becomes better for each principle component at each fit. When we then use cross validation when fitting the logistic regression, the second plot shows a more accurate distribution of the classificatio accuracy score for each principle components. However we can't refuse that both classifiers might overfit, which is also indicated by the decrease of accuracy in the cross validation plot. So around principle component 50 the decrease of accuracy could indicate a overfit. We can also see how the y axe changes radically in the two blots: plot 1 ranges from around 0.27 to 0.63, while in plot 2 it ranges from 0.23 to 0.37.

iii. What is the number of principal components, $k_{max\_accuracy}$, that results in the greatest classification accuracy when cross-validated?

In [ ]:
```
max_score_pc = np.unravel_index(np.argmax(cv_score_array), cv_score_array.sha
print("the number of principle component with the highest classification accu
```

```
the number of principle component with the highest classification accuracy sco
re is: 16 where the value is: 36.88291139240506 %
```

iv. How many percentage points is the classification accuracy increased with relative to the to the full-dimensional, d, dataset

In [ ]:
```
## to work with the full data, we should prepare it. It is already equalized:
print(data_equal.shape)
```

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

In [ ]:
```
#this should then be reshaped, to be worked with:
data_equal_reshape = data_equal.reshape(396, -1)
print(data_equal_reshape.shape)
```

```
(396, 25602)
```

In [ ]:
```
logR_d = LogisticRegression(penalty='none', solver = 'newton-cg', random_stat
logR_d.fit(data_equal_reshape[:,max_score_pc],y_equal)
```

```
scores_d = cross_val_score(logR_d, data_equal_reshape[:,max_score_pc],y_equal
mean_cv_score_d = np.mean(scores_d)

print("the accuracy score of the principle component with the highest previou
```

```
the accuracy score of the principle component with the highest previously (num
ber 16), has in the full data the accuracy score: 24.240506329113927 %
```

In [ ]:

```
#the difference between these two in percent:
difference_percent = ((np.max(cv_score_array)-(mean_cv_score_d))/(mean_cv_sco
print("the increase in percent is:", difference_percent, "%")

#the difference in percent point:
dif_point = ((np.max(cv_score_array))-(mean_cv_score_d))*100
print("the increase in percent point:", dif_point,"points")
```

```
the increase in percent is: 52.15404699738902 %
the increase in percent point: 12.642405063291134 points
```

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 the first exercise (2.1) logistic regression with in-sample validation is used, leaving both our train and test data unchanged throughout the modelling. This gives a model that only gets really good at predicting on that specific test set, which is what we see in the first plot, and this will then perform worse when predicting unseen new data. The criteria is here the best accuracy score. In comparison, we use cross-validation with 5 folds in exercise 2.2, which means the whole data set is split into train and test sets through 5 iterations and get a performance measure (classification accuracy score) for each fold. The mean performance is taken to evaluate our model, and in this way we will become better at predicting unseen data. The criteria is in this case the cross validation score. I'd argue that a good model is a generalisable model, which has a high classification accuracy on unseen data - which makes the cross validated the better choice in this case. This model is also opposite the logistic regression with no cross validation more proun not to overfit, which is an advantage.

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`.

In [ ]:

```
from sklearn.decomposition import PCA
#fit finding eigenvector values, and then transform
#used like this:

#pca = PCA(n_components=2)
#principalComponents = pca.fit_transform(x)
```

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)

```
In [ ]:
#RUNNING THE ANALYSIS PCA STYLE:
cv_score_pca = []
cv = StratifiedKFold()
n_dim = pc_list[max_score_pc]

for i in range(251):
  logR_pca = LogisticRegression(penalty='none', solver = 'newton-cg', random_
  data_prep = data_equal[:,:,i] #making sure we take the data for all 251 tim
  data_prep = sc.fit_transform(data_prep) #standardizing the data
  pca = PCA(n_components=n_dim) #making the principle components on my best P
  X_pca = pca.fit_transform(data_prep) #fitting it - this is now the X matrix
  logR_pca.fit(X_pca, y_equal) #fitting the new X to the log regression
  scores_pca = cross_val_score(logR_pca, X_pca ,y_equal, cv = 5 ) #finding th
  mean_cv_score_pca = np.mean(scores_pca)
  cv_score_pca.append(mean_cv_score_pca)

cv_score_array_pca = np.array(cv_score_pca)
```

```
In [ ]:
#max_pca = np.unravel_index(np.argmax(cv_score_array_pca), cv_score_array_pca
print("the accuracy score over time for the PCA:", cv_score_array_pca)
```

```
the accuracy score over time for the PCA: [0.27025316 0.28287975 0.24990506 0.
27259494 0.26       0.26756329
 0.25759494 0.2525     0.24993671 0.26746835 0.23734177 0.2171519
 0.23231013 0.26756329 0.28037975 0.27022152 0.29044304 0.26012658
 0.26278481 0.24243671 0.24487342 0.27512658 0.2625     0.23984177
 0.2525     0.26265823 0.29291139 0.29560127 0.29044304 0.30822785
 0.29813291 0.29541139 0.2625     0.23731013 0.23743671 0.22724684
 0.23227848 0.26756329 0.27015823 0.27025316 0.30063291 0.30822785
 0.29047468 0.25006329 0.25268987 0.24762658 0.25262658 0.245
 0.23996835 0.24487342 0.25737342 0.25987342 0.27746835 0.26753165
 0.27768987 0.26512658 0.2828481  0.26765823 0.24493671 0.29291139
 0.2928481  0.23974684 0.20693038 0.22455696 0.26234177 0.24981013
 0.24246835 0.27515823 0.29272152 0.30044304 0.25762658 0.245
 0.25256329 0.245      0.25246835 0.24737342 0.23990506 0.26256329
 0.24987342 0.23471519 0.27512658 0.25240506 0.29281646 0.30297468
 0.30047468 0.28781646 0.28525316 0.28522152 0.26493671 0.28781646
 0.28547468 0.2981962  0.29800633 0.32072785 0.31325949 0.30060127
 0.30050633 0.29547468 0.29794304 0.31306962 0.31797468 0.28515823
 0.2978481  0.29294304 0.28044304 0.30560127 0.33085443 0.33591772
 0.33091772 0.33585443 0.34848101 0.36370253 0.36882911 0.33844937
 0.34107595 0.33088608 0.29303797 0.24993671 0.27259494 0.26506329
 0.24237342 0.27265823 0.26753165 0.28512658 0.26240506 0.25987342
 0.27262658 0.29028481 0.28528481 0.31306962 0.30041139 0.2928481
 0.29022152 0.31041139 0.29037975 0.26509494 0.28015823 0.27765823
 0.27509494 0.27259494 0.28009494 0.25487342 0.29544304 0.31313291
 0.30291139 0.29537975 0.29287975 0.28528481 0.26246835 0.25490506
 0.24981013 0.27003165 0.27259494 0.25993671 0.29278481 0.28787975
 0.32579114 0.29553797 0.28541139 0.26506329 0.26253165 0.28528481
 0.30294304 0.29791139 0.27765823 0.27759494 0.25737342 0.27253165
 0.27509494 0.26509494 0.26246835 0.25493671 0.27515823 0.30291139
 0.30275316 0.28262658 0.29791139 0.28281646 0.29310127 0.30056962
 0.29797468 0.28025316 0.28525316 0.29810127 0.27012658 0.27765823
 0.27015823 0.26503165 0.27012658 0.27265823 0.25996835 0.27265823
 0.245      0.22731013 0.24746835 0.28028481 0.27787975 0.28541139
 0.27528481 0.28028481 0.28797468 0.30294304 0.29794304 0.30056962
 0.29813291 0.29800633 0.27272152 0.27272152 0.29294304 0.32060127
 0.31810127 0.30300633 0.2903481  0.28031646 0.27775316 0.24987342
 0.24487342 0.24243671 0.25259494 0.25506329 0.24243671 0.27012658
 0.26759494 0.29037975 0.29291139 0.29050633 0.30572785 0.31832278
 0.32588608 0.30053797 0.32316456 0.30556962 0.2803481  0.28787975
 0.29044304 0.31060127 0.3256962  0.30550633 0.31060127 0.31803797
 0.33063291 0.32547468 0.31037975 0.29528481 0.29281646 0.27259494
 0.26006329 0.24993671 0.27772152 0.29531646 0.29525316]
```

In [ ]:

```
import warnings
warnings.filterwarnings(action='once')

# THE SECOND ANALYSIS NO PCA, ALL DATA:
all_data_score = []
cv = StratifiedKFold()
for i in range(251):
  logR_= LogisticRegression(penalty='none', solver = 'newton-cg', random_stat
  data_prep_ = data_equal[:,:,i] #making sure we take the data for all 251 ti
  data_prep_ = sc.fit_transform(data_prep_) #standardizing the data
  logR_.fit(data_prep_, y_equal) #fitting the new X to the log regression
  scores_ = cross_val_score(logR_, data_prep_ ,y_equal, cv = cv ) #finding th
  all_scores = np.mean(scores_)
  all_data_score.append(all_scores)
```

```
/Users/laura/opt/miniconda3/envs/methods3/lib/python3.9/site-packages/sklearn/
utils/optimize.py:210: ConvergenceWarning: newton-cg failed to converge. Incre
ase the number of iterations.
  warnings.warn(
/Users/laura/opt/miniconda3/envs/methods3/lib/python3.9/site-packages/sklearn/
utils/optimize.py:210: ConvergenceWarning: newton-cg failed to converge. Incre
ase the number of iterations.
  warnings.warn(
/Users/laura/opt/miniconda3/envs/methods3/lib/python3.9/site-packages/sklearn/
utils/optimize.py:210: ConvergenceWarning: newton-cg failed to converge. Incre
ase the number of iterations.
  warnings.warn(
/Users/laura/opt/miniconda3/envs/methods3/lib/python3.9/site-packages/sklearn/
utils/optimize.py:210: ConvergenceWarning: newton-cg failed to converge. Incre
ase the number of iterations.
  warnings.warn(
/Users/laura/opt/miniconda3/envs/methods3/lib/python3.9/site-packages/sklearn/
utils/optimize.py:210: ConvergenceWarning: newton-cg failed to converge. Incre
ase the number of iterations.
  warnings.warn(
/Users/laura/opt/miniconda3/envs/methods3/lib/python3.9/site-packages/sklearn/
utils/optimize.py:210: ConvergenceWarning: newton-cg failed to converge. Incre
ase the number of iterations.
  warnings.warn(
```

In [ ]:

```
print("the accuracy score over time for the data without pca:", all_data_scor
```

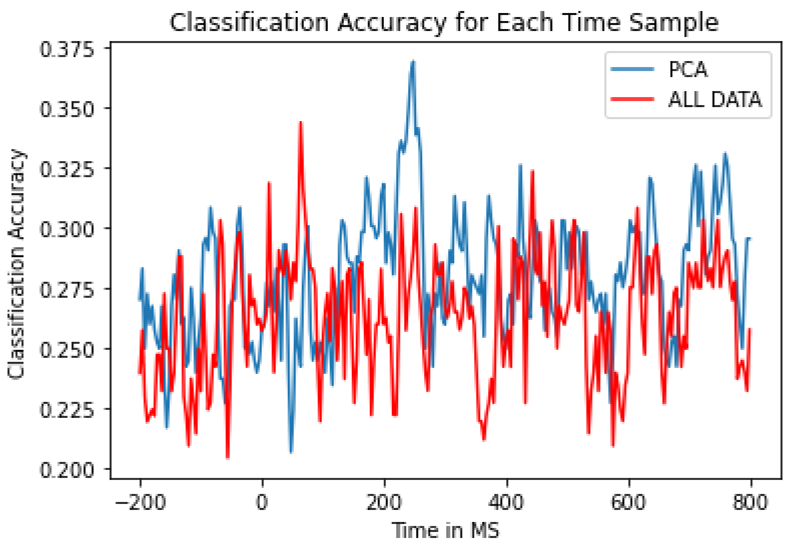
```
the accuracy score over time for the data without pca: [0.23977848101265825,
0.2575, 0.22974683544303795, 0.2195253164556962, 0.22202531645569618, 0.224556
96202531644, 0.22199367088607597, 0.2473101265822785, 0.24740506329113923, 0.2
3221518987341772, 0.2727215189873418, 0.24996835443037976, 0.2499050632911392,
0.23218354430379745, 0.23981012658227846, 0.2652531645569621, 0.28791139240506
325, 0.2880379746835443, 0.22993670886075948, 0.22224683544303797, 0.209556962
02531646, 0.23740506329113925, 0.22740506329113921, 0.21458860759493673, 0.249
74683544303797, 0.23218354430379745, 0.2725316455696203, 0.24740506329113923,
0.22465189873417724, 0.2273417721518987, 0.24737341772151894, 0.24237341772151
896, 0.27262658227848097, 0.3029746835443038, 0.2929113924050633, 0.2524050632
911392, 0.20455696202531642, 0.23474683544303793, 0.272879746835443, 0.2829113
924050633, 0.2954430379746835, 0.298006329113924, 0.27281645569620255, 0.25765
82278481013, 0.24246835443037976, 0.2802848101265823, 0.26772151898734176, 0.2
7018987341772155, 0.25999999999999995, 0.26231012658227854, 0.257246835443038,
0.2598101265822785, 0.2651582278481013, 0.31835443037974687, 0.270189873417721
55, 0.24, 0.2601582278481013, 0.2905379746835443, 0.28284810126582277, 0.28018
98734177215, 0.290379746835443, 0.28284810126582277, 0.27022151898734176, 0.28
53481012658228, 0.27775316455696203, 0.30291139240506326, 0.3435443037974684,
0.31585443037974686, 0.3031645569620253, 0.29060126582278484, 0.28268987341772
156, 0.28268987341772156, 0.2752215189873418, 0.24240506329113926, 0.219683544
3037975, 0.25240506329113926, 0.2650632911392405, 0.27281645569620255, 0.25284
81012658228, 0.2830696202531645, 0.2751898734177215, 0.24496835443037973, 0.26
272151898734175, 0.27778481012658224, 0.2372784810126582, 0.28025316455696203,
0.28287974683544304, 0.255126582278481, 0.22727848101265824, 0.244905063291139
23, 0.2827848101265823, 0.2854113924050633, 0.2650316455696203, 0.247341772151
89873, 0.2702848101265823, 0.2222151898734177, 0.24496835443037973, 0.26003164
55696202, 0.26006329113924054, 0.2827215189873418, 0.2600316455696202, 0.26253
16455696202, 0.2524367088607595, 0.25506329113924053, 0.2224367088607595, 0.22
224683544303797, 0.25756329113924054, 0.30550632911392406, 0.2877848101265822
5, 0.2574367088607595, 0.27259493670886076, 0.2802215189873417, 0.290316455696
2025, 0.3081012658227848, 0.28034810126582277, 0.26765822784810134, 0.25237341
7721519, 0.2399050632911392, 0.232373417721519, 0.2626582278481013, 0.26768987
341772155, 0.29287974683544304, 0.2802848101265823, 0.2804430379746835, 0.2854
746835443038, 0.2676898734177215, 0.2626898734177215, 0.2677848101265823, 0.27
772151898734176, 0.2649683544303797, 0.2651582278481012, 0.25775316455696207,
0.2625632911392405, 0.275126582278481, 0.27275316455696197, 0.2624683544303797
5, 0.26746835443037975, 0.2599367088607595, 0.23981012658227846, 0.21971518987
34177, 0.2195886075949367, 0.21202531645569622, 0.2222151898734177, 0.22727848
101265824, 0.23734177215189875, 0.2273417721518987, 0.2601582278481013, 0.3004
7468354430384, 0.260126582278481, 0.2424050632911392, 0.25259493670886074, 0.2
575, 0.24224683544303796, 0.2952848101265823, 0.29287974683544304, 0.270253164
556962, 0.2877848101265823, 0.2853164556962025, 0.22727848101265824, 0.2802848
101265823, 0.2979430379746835, 0.3233227848101266, 0.2829746835443038, 0.28034
810126582277, 0.29787974683544305, 0.267626582278481, 0.2774683544303797, 0.25
477848101265826, 0.28522151898734177, 0.30287974683544305, 0.2903797468354430
4, 0.2500316455696202, 0.2675632911392405, 0.2624367088607594, 0.2599050632911
392, 0.2651582278481013, 0.2700316455696202, 0.2954746835443038, 0.30303797468
35443, 0.26778481012658223, 0.2651898734177215, 0.28018987341772156, 0.2979113
9240506326, 0.24237341772151896, 0.21468354430379746, 0.23227848101265822, 0.2
39873417721519, 0.25509493670886074, 0.23224683544303798, 0.2651582278481012,
0.2575632911392405, 0.239873417721519, 0.265, 0.2574050632911392, 0.2095569620
2531646, 0.23987341772151902, 0.23481012658227848, 0.2246835443037975, 0.21977
848101265823, 0.23468354430379748, 0.23974683544303796, 0.27531645569620256,
0.2752215189873418, 0.2904113924050633, 0.3081012658227848, 0.282689873417721
5, 0.2601582278481013, 0.2475, 0.2877215189873418, 0.2876898734177215, 0.27259
493670886076, 0.2902848101265823, 0.29287974683544304, 0.275253164556962, 0.23
99367088607595, 0.22712025316455695, 0.24981012658227847, 0.2650316455696202,
0.2550632911392405, 0.27281645569620255, 0.2751898734177215, 0.257310126582278
5, 0.24227848101265823, 0.255, 0.249873417721519, 0.285253164556962, 0.2802531
6455696203, 0.27512658227848097, 0.28525316455696204, 0.275253164556962, 0.275
1898734177215, 0.3031645569620253, 0.28537974683544304, 0.2778797468354431, 0.
2830379746835443, 0.275253164556962, 0.29287974683544304, 0.30306962025316453,
0.27537974683544303, 0.28300632911392404, 0.28800632911392404, 0.2904430379746
835, 0.2778164556962025, 0.2700316455696202, 0.2776898734177215, 0.23734177215
189875, 0.24243670886075952, 0.24493670886075952, 0.24000000000000005, 0.23234
177215189872, 0.2576898734177215]
```

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*-axis

In [ ]:

```
plt.plot(times, cv_score_array_pca)
plt.plot(times, all_data_score, color = 'red')
plt.title("Classification Accuracy for Each Time Sample")
plt.xlabel("Time in MS")
plt.ylabel("Classification Accuracy")
plt.legend(['PCA', 'ALL DATA'])
```

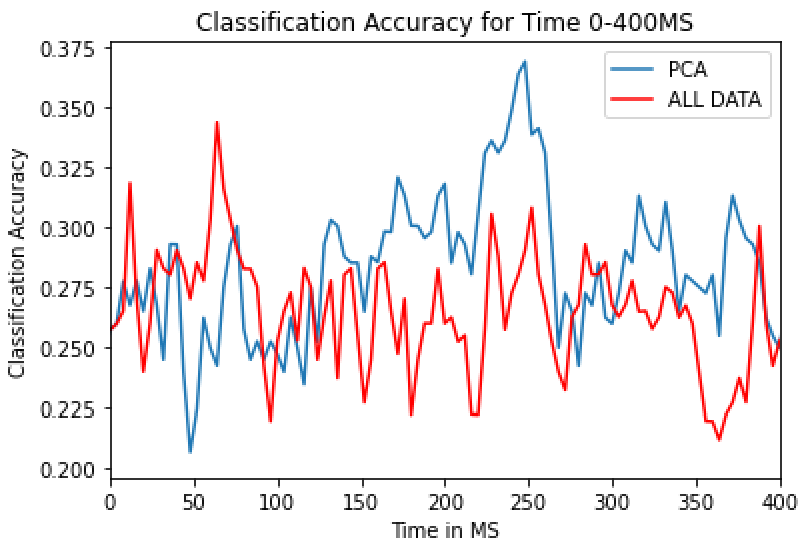
Out[ ]:
```
<matplotlib.legend.Legend at 0x159abcf70>
```



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

In [ ]:
```
plt.plot(times, cv_score_array_pca)
plt.plot(times, all_data_score, color = 'red')
plt.xlim((0, 400))
plt.title("Classification Accuracy for Time 0-400MS")
plt.xlabel("Time in MS")
plt.ylabel("Classification Accuracy")
plt.legend(['PCA', 'ALL DATA'])
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x159b8cc40>
```



The plot shows the average of all the accuracy scores of all times classified by either PCA or the regression on all data. We see a peak in the accuracy of the PCA around 250 and a general tendency of it performing better when classifying. This makes sense explained by the sensors must have a very high activety in this. The higher the sensor activation the higher the covariance, which leads to the better performance of the PCA. When the *covariane isn't that high, the difference of the classifiers aren't that big, because the new fitted PCA's doesn't provide any new information.*

The difference between the two analysis

The main difference between the two analysis is the number of principle components used when classifying. Inspecting the third plot the first 16 components results in the greatest classification accuracy, this being 36%. The total number of principle components (102) performs a bit worse, its maximum being 34%. Even though the 16 principle components seems to performe better, there is only a small differnence. The performance of the model is better on this PCA reduced dataset (16 principle components), which can be explained by the reduction of covariated variables, that did not provide information to the classification. Hence type of analysis ensures that the data does not contain any dirsturbing noise (remaining from the princple components), but only the information that contributes to the classification.