Practical exercise 8 , Methods 3 2021, autumn semester

Emma Risgaard Olsen 🤠

Exercises and objectives

1) Load the magnetoencephalographic recordings and do some initial plots to understand the data

2) Do logistic regression to classify pairs of PAS-ratings

3) Do a Support Vector Machine Classification on all four PAS-ratings

Loading packages

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
plt.rcParams['figure.dpi']= 300
```

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

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 [3]:
```
data = np.load("/Users/emmaolsen/OneDrive - Aarhus Universitet/UNI/Methods3

y = np.load("/Users/emmaolsen/OneDrive - Aarhus Universitet/UNI/Methods3/g
```

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

In [4]:
```
def equalize_targets(data, y):
    np.random.seed(7)
    targets = np.unique(y) ## find the number of targets
    counts = list()
    indices = list()
    for target in targets:
        counts.append(np.sum(y == target)) ## find the number of each targ
        indices.append(np.where(y == target)[0]) ## find their indices
    min_count = np.min(counts)
    # randomly choose trials
    first_choice = np.random.choice(indices[0], size=min_count, replace=Fa
    second_choice = np.random.choice(indices[1], size=min_count,replace=Fa
    third_choice = np.random.choice(indices[2], size=min_count, replace=Fa
    fourth_choice = np.random.choice(indices[3], size=min_count, replace=F

    # create the new data sets
    new_indices = np.concatenate((first_choice, second_choice, third_choic
    new_y = y[new_indices]
    new_data = data[new_indices, :, :]

    return new_data, new_y
```

In [5]:
```
# Equalizing targets
data_eq, y_eq = 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) iv. Scale the data using `StandardScaler`

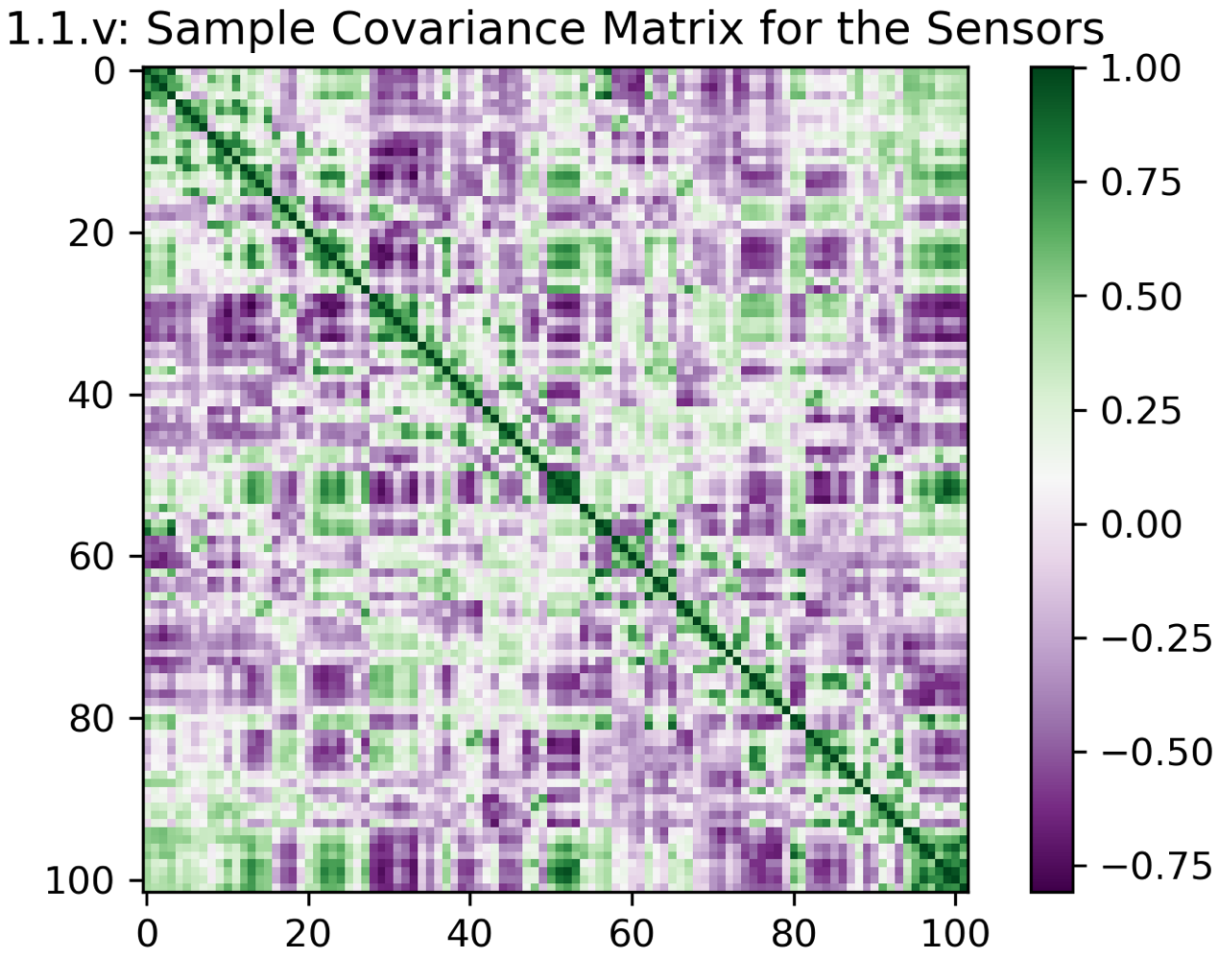
In [6]:
```
# Generating numbers from -200 to 800 in intervals of 4. Start from 200, e
times = np.arange(-200, 804,4)

timeindex248 = np.where(times==248)
timeindex248 # The time index corresponding to 248 ms (max avr response) i


# Reduce the dimensionality of data
data_eq_red = data_eq[:,:,112]
```

iv. Scale the data using `StandardScaler`

In [7]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

data_sc = sc.fit_transform(data_eq_red)
```

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 [20]:
```
cov_mat = np.cov(data_sc.T)

plt.figure()
plt.imshow(cov_mat, cmap = "PRGn")
plt.colorbar()
plt.title("1.1.v: Sample Covariance Matrix for the 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 indicates that there is not complete independence of the signals measured by the 102 sensors. It seems that some sensors pick up the same signal. Had there been complete independence, the off-diagonal would show 0 covariance, i.e. a light green/purple/almost white colour, which is not the case. We see a lot of dark purple and green colours, indicating that the signals aren't independent of each other.

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

In [59]:
```
print("Matrix rank = ", np.linalg.matrix_rank(cov_mat))
```

```
Matrix rank =  97
```

I get the integer value 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 [47]:
```
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)

print(eigen_vals.shape)
print(eigen_vecs.shape)

print('\nEigenvalues \n%s' % eigen_vals)
```

```
(102,)
(102, 102)

Eigenvalues
[ 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  5.38232974e-02  5.30717201e-02
  5.52056614e-02  1.89773602e-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  2.47132649e-02
  3.90017448e-02  2.84125782e-02  3.36438792e-02  4.07416727e-02
  4.14389338e-02  2.95657187e-02  3.99926065e-02  3.44327716e-02
  2.49751605e-02  5.47188291e-16  3.31455092e-17 -4.77163425e-16
 -3.16830041e-16 -2.26629244e-16]
```

In [48]:
```
# Looking at the real parts of the eigenvectors and -values
eigen_vals = np.real(eigen_vals)
eigen_vecs = np.real(eigen_vecs)
```

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 [49]:
```
eigen_vals_abs = np.abs(eigen_vals)
```

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.

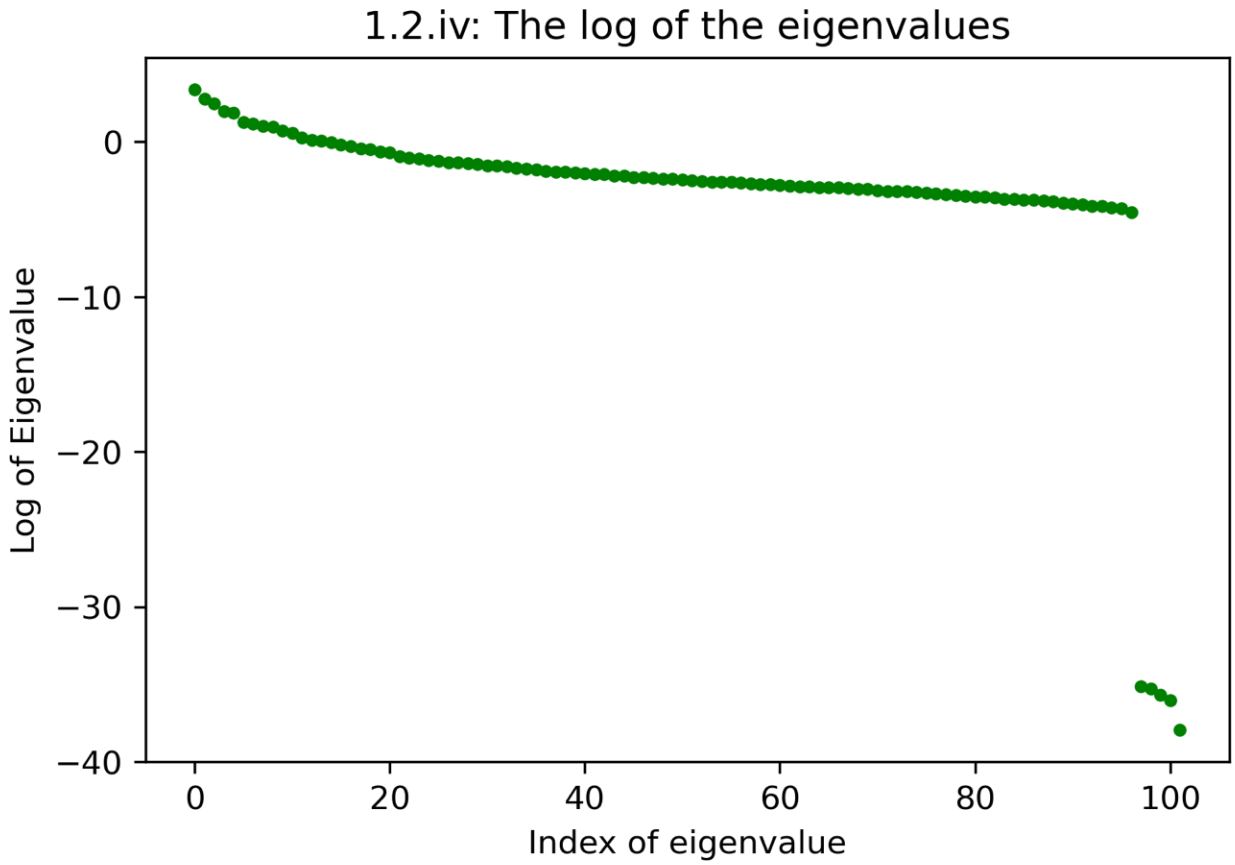
In [50]:
```
sorted_indices = eigen_vals_abs.argsort()
sorted_indices = np.flip(sorted_indices)
```

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]`

In [51]:
```
eigen_vals = eigen_vals_abs[sorted_indices]
eigen_vecs = eigen_vecs[:,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 [52]:
```
plt.figure()
plt.plot(np.log(eigen_vals), '.', color="green")
plt.title("1.2.iv: The log of the eigenvalues")
plt.xlabel("Index of eigenvalue")
plt.ylabel("Log of Eigenvalue")
plt.show()
```



There seem to be five values (the 5 with the highest index) that stand out from the rest. The rest have an log value of between -10 and 0, whereas the 5 outliers have a log value of between -30 and -40.

The integer value for the matrix rank was 97. As the rank of a matrix is the maximum number of its linearly independent column or vectors, this makes sense. We have 102 dimensions (sensors), and the 5 outliers from the above plot constitutes the noise that explains the matrix rank of 97.

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

In [53]:
```
W = eigen_vecs # I already used the command: eigen_vecs = eigen_vecs[:,sor
```

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 [54]:
```
Z = data_sc @ W
```

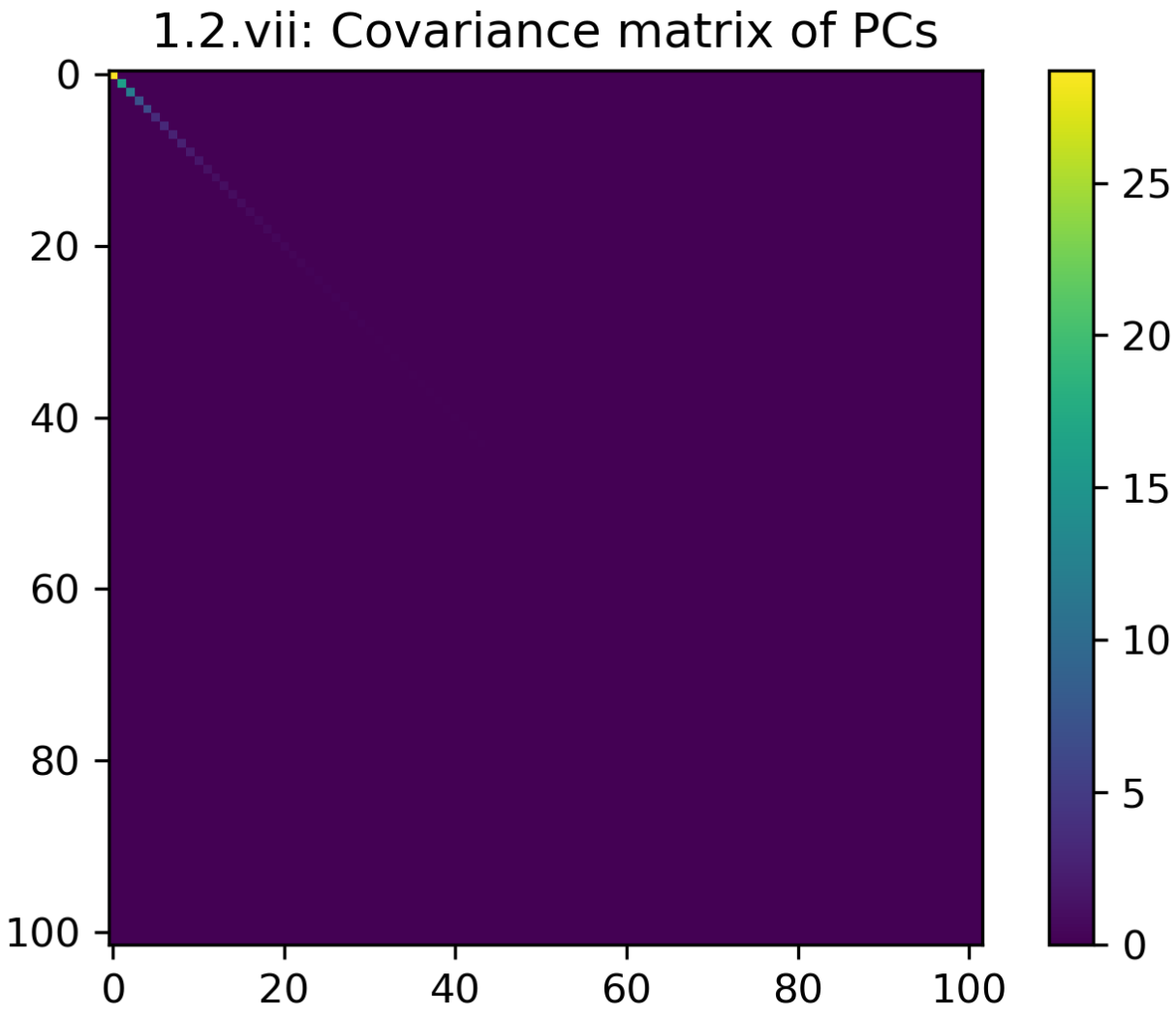
In [55]:
```
checkcorrect = np.isclose(data_sc,Z@W.T) # checking that I did everything
n_false = np.sum(checkcorrect == False)
print("Values of X not equal to our original X: {}".format(n_false))
```

```
Values of X not equal to our original X: 0
```

vii. Create a new covariance matrix of the principal components (n=102) - plot it! What has happened off-diagonal and why?

In [56]:
```
cov_matrix = np.cov(Z, rowvar = False) # rowvar = false, choosing 2nd dime
```

In [57]:
```
plt.figure()
plt.imshow(cov_matrix)
plt.colorbar()
plt.title("1.2.vii: Covariance matrix of PCs")
plt.show()
```



The covariance matrix was computed to identify correlations. We compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components.

The covariance matrix computation was done with the aim of understanding how the variables in our data set vary from the mean with respech to each other. It allows us to assess whether there is a relationship between them. If the variables are highly correlated, they might contain redundant informaiton.

The diagonal shows the covariance matrix of PCs. The PCs are sorted/indexed so that the one that maximizes the variance of the projected data (i.e., has the highest explained variance ratio)comes first (is the first pricipal component). We can also see in the diagonal, that the first PCs are the ones that covary more with themselves. In the main diagonal, we have the variances of each initial variable in the main diagonal.

The off-diagonal shows no covariance which makes sense as the principal components are mutually orthogonal.

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 $Z_{d\times k}$ and $Z_{n\times k}$ y (the target vector). Fit (`.fit`) 102 models based on: $k = [1, 2, \ldots, 101, 102]$ and $d = 102$. For each fit get the classification accuracy, (`.score`), when applied to $Z_{d\times k}$ and $Z_{n\times k}$ and y. This is an in-sample validation. Use the solver `newton-cg` if the default solver doesn't converge

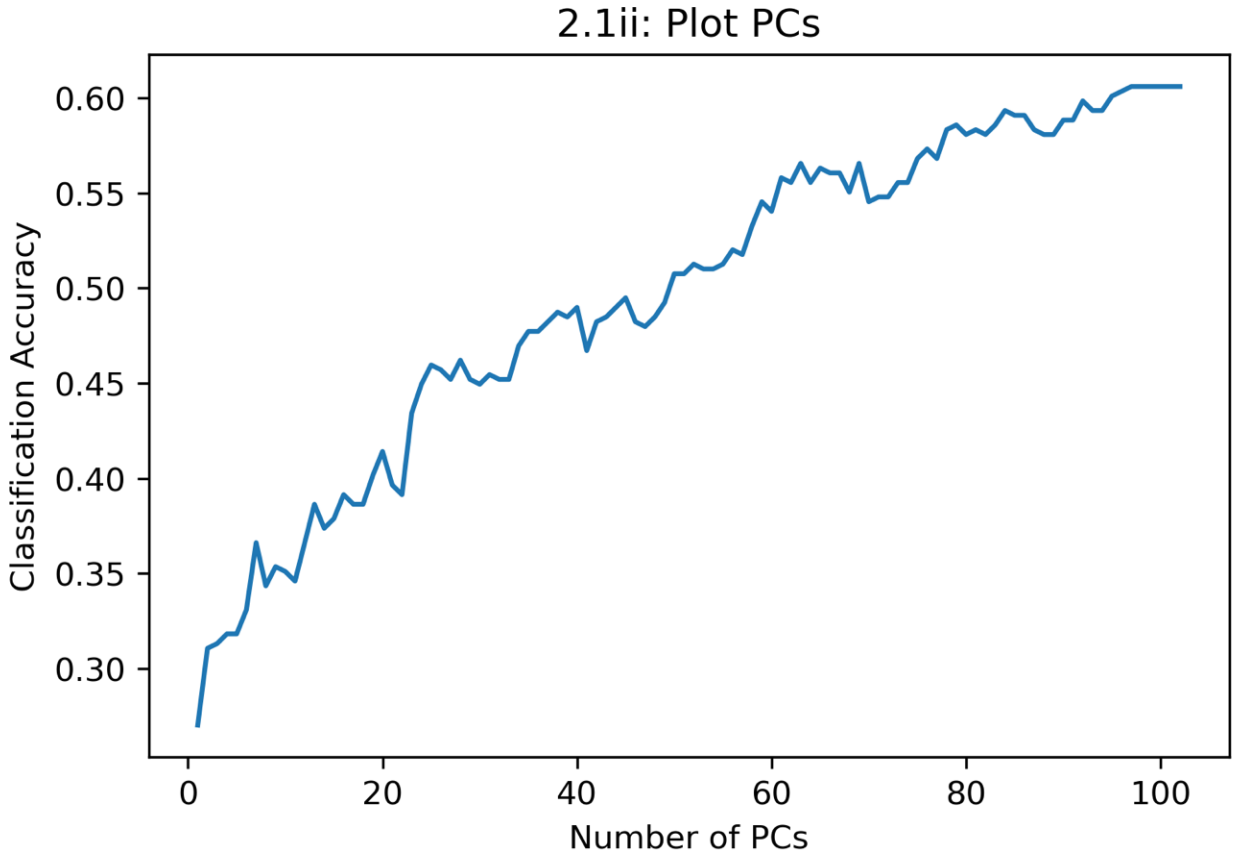
In [61]:
```
from sklearn.linear_model import LogisticRegression
```

In [192…
```
lr_scores = []
n_pc = np.arange(1, 103) # number of principle components

for i in n_pc:
  lr = LogisticRegression(solver = "newton-cg")
  lr.fit(Z[:,:i].reshape(-1, i), y_eq) # create fits with increasing amoun
  lr_scores.append(lr.score(Z[:,:i].reshape(-1, i), y_eq))
```

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 [193…

```
plt.figure()
plt.plot(n_pc, lr_scores, "-")
plt.title("2.1ii: Plot PCs")
plt.xlabel("Number of PCs")
plt.ylabel("Classification Accuracy")
plt.show()
```



The classification accuracy increases with the number of PCs. This is because the model is allowed to fit more noise (the more PCs are used), resulting in overfitting.

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

There seems to be no effect of adding the five last component (the graph flattens out).

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

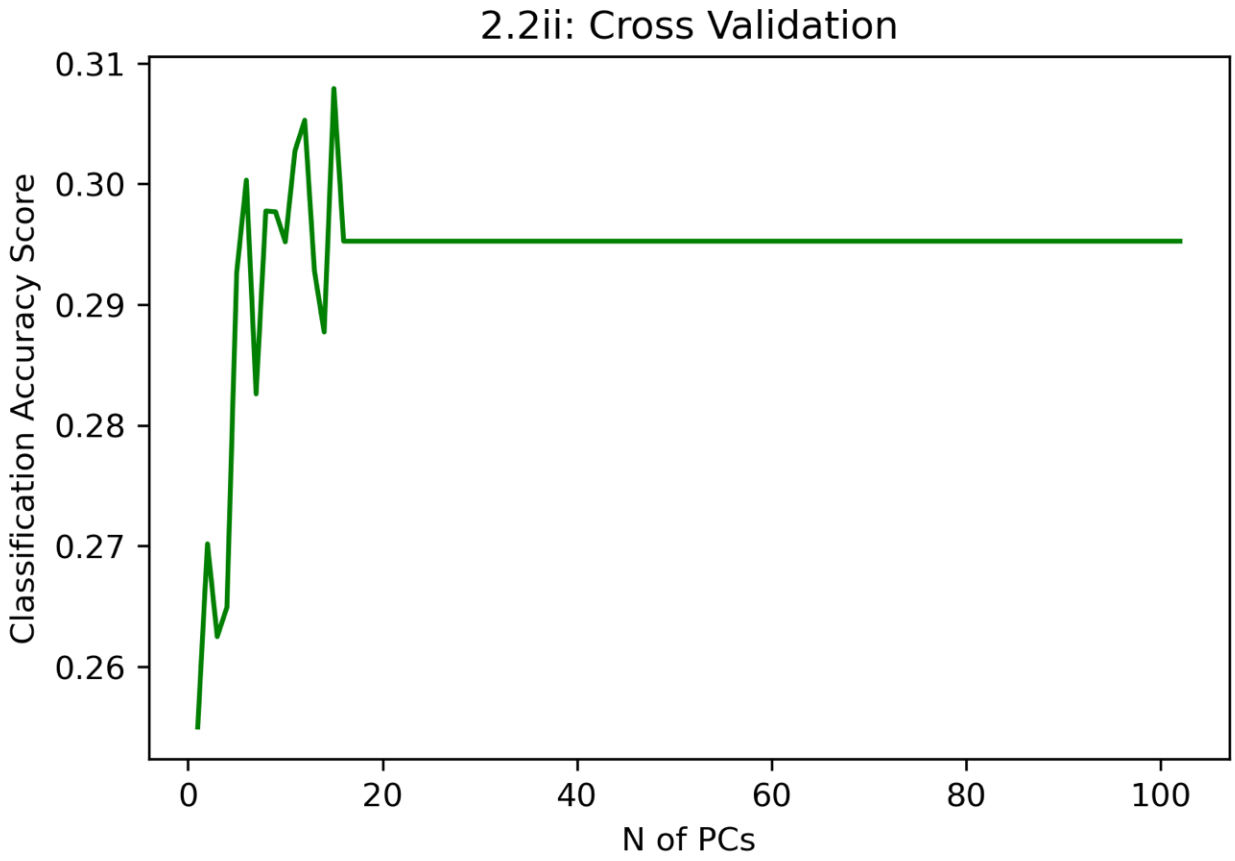
In [140…

```
logReg = LogisticRegression(penalty ='none', solver= 'newton-cg', random_st

# For cross validation
from sklearn.model_selection import StratifiedKFold, cross_val_score
cv = StratifiedKFold(n_splits=5)

# Empty array to append later
cv_scores = []

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

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 [141…

```
plt.figure()
plt.plot(n_pc, cv_scores, "-", color="green")
plt.title("2.2ii: Cross Validation")
plt.xlabel("N of PCs")
plt.ylabel("Classification Accuracy Score")
plt.show()
```



We now see that the classification accuracy score doesn't just increase when increasing the number of Principal Components. This makes sense that when doing cross-validation, we avoid fitting noise as we use both training and test data

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

In [82]:
```
indexmax= np.argmax(cv_scores)
indexmax
```

Out[82]:
```
15
```

The index indicating the number of principal components resulting in the greatest classification accuracy when cross-validated is 15. As python starts indexation from 0, the number of PCs resulting in the max classification accuracy is 16. This seems to fit when you eyeball the plot above.

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

In [83]:
```
print("Classification accuracy is increased with", round(cv_scores[indexmax
```

```
Classification accuracy is increased with 7.9 Percentage Points relative to
the full-dimensional, d, dataset
```

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 used the same data for training and testing. This allows for a very great model fit, but with a risk of lacking generalisability. In exercise 2.2, we used mused 5k-cross-validation which allowed us to use different subsets of the data for training and testing. This makes our model fit our data a little worse, but increases generalisability, i.e. the model will perform better at unseen data.

In other words, the optimization criteria for the two exercises are different.

We wanna find a sweet spot where we can make the model fit our data well, but without fitting noise - while also making the model generalisable. Our performance criteria in machine learning is making the model able to predict unseen data, so one would think cross-validation is the better approach.

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 [84]:
```
from sklearn.decomposition import PCA
```

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)

First) Reduce dimensionality to $k_{max_accuracy}$ dimensions using PCA

Our $k_{max_accuracy}$ is 16

In [144…

```
from sklearn.decomposition import PCA

cv = StratifiedKFold(n_splits=5)
pca_kmax = PCA(n_components=16)

pca_kmax_scores = []

for i in range(251):

  lr = LogisticRegression(solver = "newton-cg")
  scaler = StandardScaler()


  X = data_eq[:,:,i] # scaling
  X_scaled = scaler.fit_transform(X)


  # Analysis w. kmax
  Z = pca_kmax.fit_transform(X_scaled)
  cv_score = cross_val_score(lr, Z, y_eq, cv = cv)
  pca_kmax_scores.append(np.mean(cv_score))
```

Second) Use the full data

In [145…

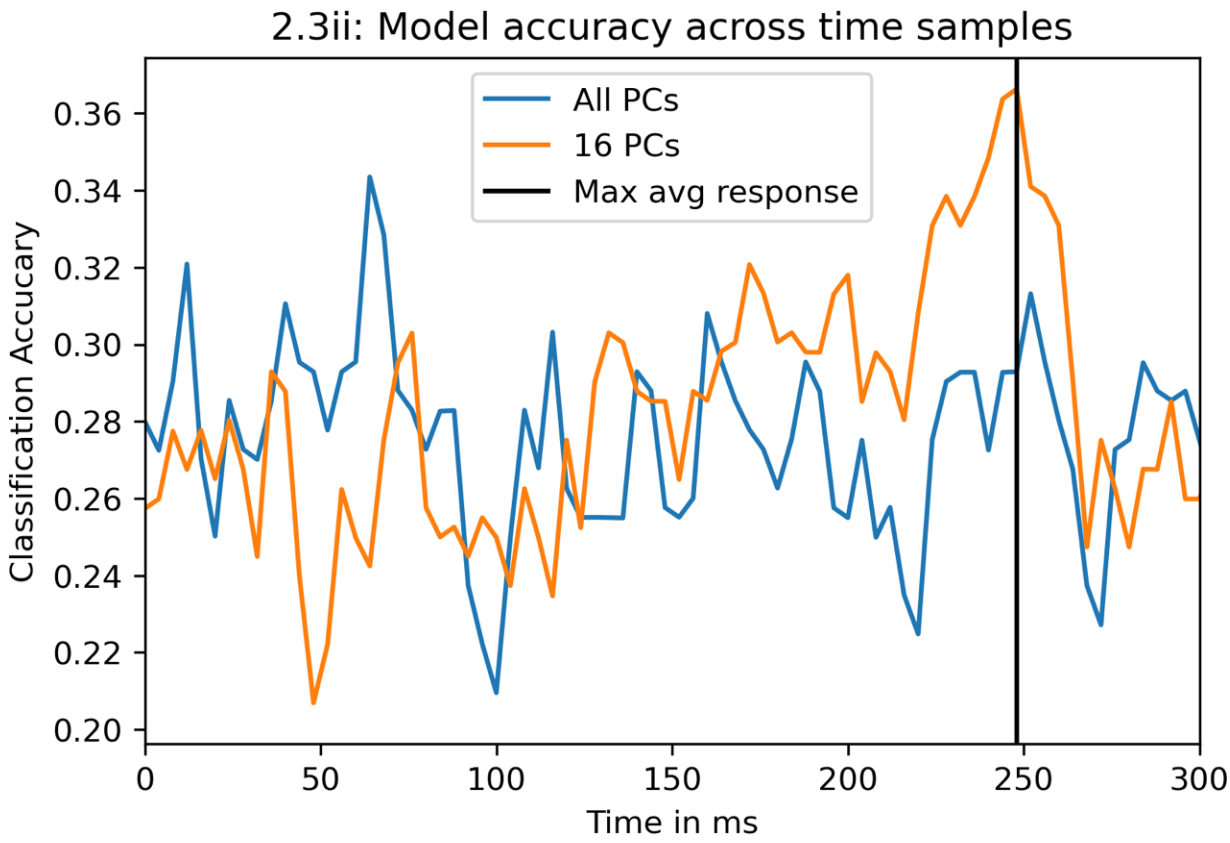
```
pca_all = PCA(n_components=None)
pca_all_scores = []


for i in range(251):

  lr = LogisticRegression(solver = "newton-cg")
  scaler = StandardScaler()

  X = data_eq[:,:,i] # scaling
  X_scaled = scaler.fit_transform(X) # scaling

  ## PCA with with full data
  Z = pca_all.fit_transform(X_scaled)
  cv_score = cross_val_score(lr, Z, y_eq, cv = cv)
  pca_all_scores.append(np.mean(cv_score))
```

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 [150…
```
plt.close("all")
```

In [180…
```
plt.figure()
plt.plot(times, pca_all_scores,label = "All PCs")
plt.plot(times,pca_kmax_scores, label = "16 PCs")
plt.title("2.3ii: Model accuracy across time samples")
plt.xlabel("Time in ms")
plt.xlim([0,300])
plt.ylabel("Classification Accucary")
plt.axvline(248, color = "black", label = "Max avg response")
plt.legend()
```

Out[180…
```
<matplotlib.legend.Legend at 0x170f49fd0>
```



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

Looking at the time interval between 0 ms and 400 ms, the model using the PCA-reduced data (orange graph) set has the higher classification accuracy as compared to using all the dimensions (blue grahp). The graph reveals that using only 16 dimensions results in a better performance, especially a 248 ms which is the time with the maximal average response. At this point, the model as an accuracy of apx. 37%. It makes sense as we found in the early assignment that time point 248 was the exact time point where the average peak of magnetic activity was highest.

However, at the first peak of detected magnetic activity (around 70 ms, the model on the full data set has a higher classification accuracy.

In [ ]:

In [ ]: