Assignment 3 , Methods 3, 2024, autumn semester

2024-10-22

```
library(reticulate)
```

```
## Warning: package 'reticulate' was built under R version 4.3.3
```

```
print(conda_list()) #this should print the path to the conda environments
```

```
##            name                                        python
## 1          base                    /opt/miniconda3/bin/python
## 2             R             /opt/miniconda3/envs/R/bin/python
## 3 methods3_2024 /opt/miniconda3/envs/methods3_2024/bin/python
```

```
use_python("/opt/miniconda3/envs/methods3_2024/bin/python")
```

import the important packages

```
import sklearn as sk
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os
from os.path import join
from sklearn.model_selection import cross_val_score,StratifiedKFold, train_test_split
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer
from sklearn.metrics import confusion_matrix
```

Exercises and objectives

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

B) Do logistic regression to classify pairs of PAS-ratings and all four-PAS-ratings

REMEMBER: In your portfolio, make sure to include code that can reproduce the answers requested in the exercises below (**MAKE A KNITTED VERSION**). If it does not KNIT, it cannot be part of your portfolio

EXERCISE A - Load the magnetoencephalographic recordings and do some initial plots to understand the data

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)

Vilma

1) 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

```
#Loading the dataset
#data_path = join("D:/Vilma/UNI/3rd. Semester/Methods_3/Assignment_3", "megmag_data.npy")
#data = np.load(data_path)

file_path_data = os.path.expanduser('~/Documents/CogSci/Sem3/Methods3/megmag_data.npy')
data = np.load(file_path_data) # Loading the file path to the variable
```

i. The data is a 3-dimensional array. The first dimension is number of repetitions of a visual stimulus,the second dimension is the number of sensors that record magnetic fields (in Tesla) that stem from neurons activating in the brain, and the third dimension is the number of time samples. How many repetitions, sensors and time samples are there?

```
#Getting the dimensions of the data
num_repetitions, num_sensors, num_time_samples = data.shape
```

ii. The time range is from (and including) -200 ms to (and including) 800 ms with a sample recorded every 4 ms. At time 0, the visual stimulus was briefly presented.
Create a 1-dimensional array called ‘times‘ that represents this.

```
#Creating time array
start_time = -200 #ms
end_time = 800 #ms
sampling_interval = 4 #ms
times = np.arange(start_time, end_time + 1, sampling_interval)
```

```
print("Number of repetitions:", num_repetitions)
```

```
## Number of repetitions: 682
```

```
print("Number of sensors:", num_sensors)
```

```
## Number of sensors: 102
```

```
print("Number of time samples:", num_time_samples)
```

```
## Number of time samples: 251
```

```
print("Time array shape:", times.shape)
```

```
## Time array shape: (251,)
```

iii. Create the sensor covariance matrix Σ_{XX}: $$\Sigma_{XX} = \frac 1 N \sum_{i=1}^N XX^T$$ N is the number of repetitions and X has s rows and t columns (sensors and time), thus the shape is $X_{s\times t}$. Do the sensors pick up independent signals? (Use ‘plt.imshow‘ to plot the sensor covariance matrix)

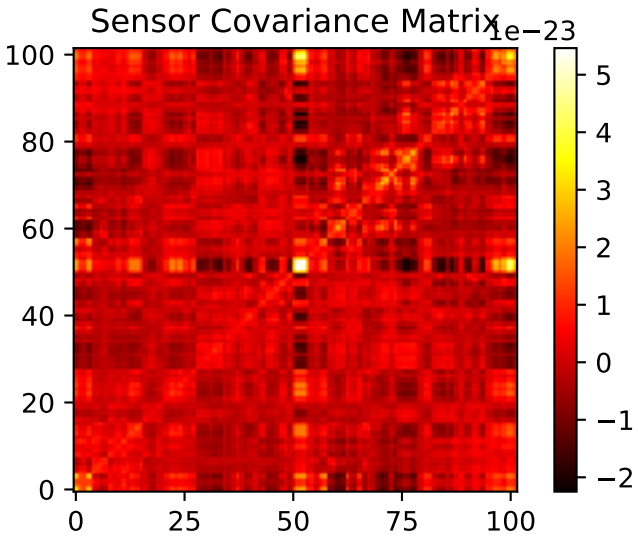
```
#N, s, t = data.shape # N = repetitions (682), s = sensors (102), t = time (251)

N = data.shape[0]
s = data.shape[1]
t = data.shape[2]

plt.figure(figsize=(5, 3))

cov_matrix_sum = np.zeros((s, s))

# looping through each trial for the repetition
for i in range(N):
    # extracting the data for the i'th trial (sensors, time)
    X = data[i, :,:]  # matrix of shape (102, 251) for trial i
    # covariance matrix for this trial
    cov_matrix_sum += np.dot(X, X.T)  # X * X^T provides a (102, 102) matrix

# averaging the covariance matrix over the trials
cov_matrix = cov_matrix_sum / N

# checking covariance matrix details
print("Covariance matrix shape:", cov_matrix.shape)
```

```
## Covariance matrix shape: (102, 102)
```

```
# plotting the covariance matrix
plt.imshow(cov_matrix, cmap='hot', interpolation='nearest')
plt.gca().invert_yaxis()
plt.title("Sensor Covariance Matrix")
plt.colorbar()
```

```
## <matplotlib.colorbar.Colorbar object at 0x161551af0>
```

```
plt.show()  # display the plot
```



iv. Make an average over the repetition dimension using `np.mean` - use the `axis` argument. (The resulting array should have two dimensions with sensor as the first and time as the second)

```
#Checking the shape of the data
print("Original Data Shape:", data.shape)
```

```
## Original Data Shape: (682, 102, 251)
```

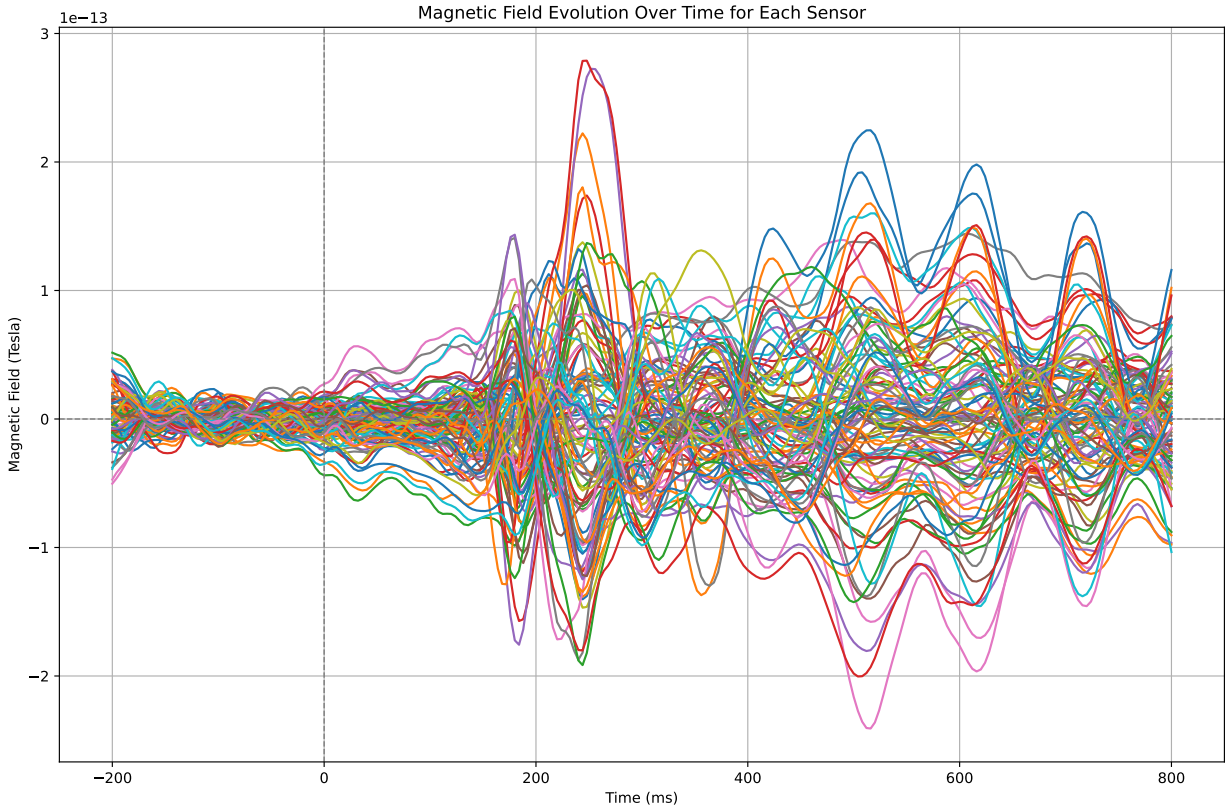
```
# Calculating the average over the repetition dimension (axis=0)
average_data = np.mean(data, axis=0)

#Checking the shape of the resulting array
print("Averaged Data Shape:", average_data.shape)
```

```
## Averaged Data Shape: (102, 251)
```

v. Plot the magnetic field (based on the average) as it evolves over time for each of the sensors (a line for each) (time on the x-axis and magnetic field on the y-axis). Add a horizontal line at $y = 0$ and a vertical line at $x = 0$ using ‘plt.axvline‘ and ‘plt.axhline‘

```
# Plotting the magnetic field for each sensor
plt.figure(figsize=(12, 8))

# Iterate over each sensor and plot
for sensor_idx in range(average_data.shape[0]):
    plt.plot(times, average_data[sensor_idx, :], label=f'Sensor {sensor_idx + 1}')

    # Adding horizontal and vertical lines
plt.axhline(0, color='gray', linestyle='--', linewidth=1)
plt.axvline(0, color='gray', linestyle='--', linewidth=1)
```

```
# Adding labels and title
plt.title('Magnetic Field Evolution Over Time for Each Sensor')
plt.xlabel('Time (ms)')
plt.ylabel('Magnetic Field (Tesla)')
plt.grid()
plt.tight_layout()  # Adjust layout to fit labels
plt.show()
```



vi. Find the maximal magnetic field in the average. Then use `np.argmax` and `np.unravel_index` to find the sensor that has the maximal magnetic field.

```
# Finding the maximal magnetic field in the average data
max_magnetic_field = np.max(average_data)

# Using np.argmax to find the index of the maximum value
max_index = np.argmax(average_data)

# Useing np.unravel_index to find the sensor and time sample corresponding to this index
sensor_idx, time_idx = np.unravel_index(max_index, average_data.shape)

# Printing the results
print("Maximal Magnetic Field:", max_magnetic_field)
```

```
## Maximal Magnetic Field: 2.7886216843591933e-13
```

```
print("Sensor Index:", sensor_idx)
```

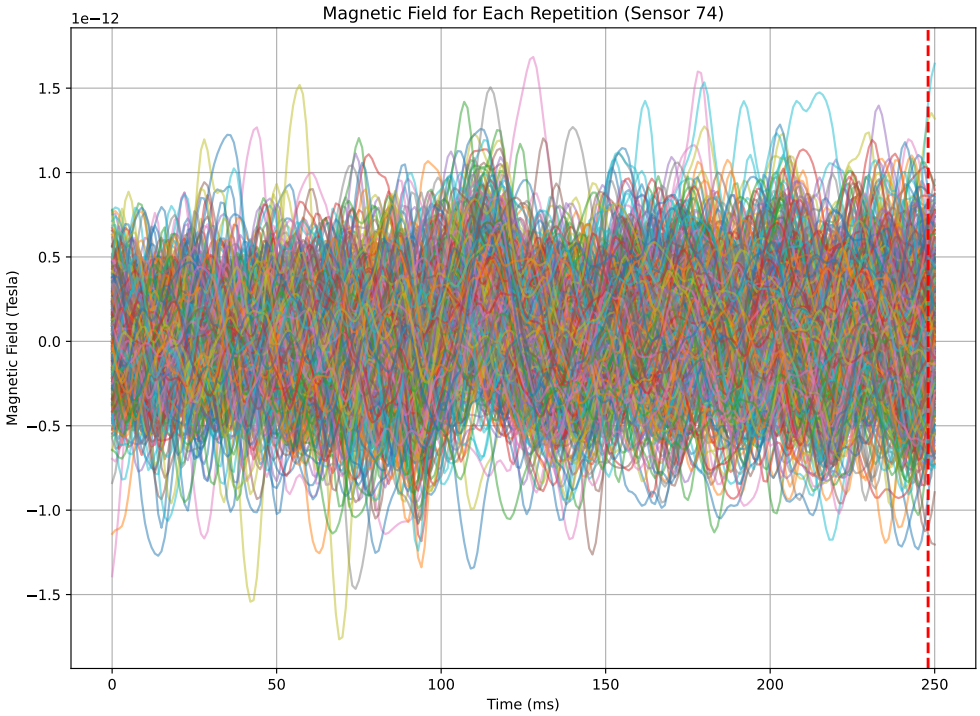
```
## Sensor Index: 73
```

```
print("Time Index:", time_idx)
```

```
## Time Index: 112
```

vii. Plot the magnetic field for each of the repetitions (a line for each) for the sensor that has the maximal magnetic field. Highlight the time point with the maximal magnetic field in the average (as found in A.1.v) using ‘plt.axvline‘

```
# Plotting the magnetic field for each repetition for the sensor with maximal
# magnetic field
plt.figure(figsize=(12, 8))

# Plotting each repetition for the sensor that has the maximal magnetic field
for repetition in range(data.shape[0]):
    plt.plot(data[repetition, sensor_idx, :], label=f'Repetition {repetition + 1}',
    alpha=0.5)

    # Highlight the time point with the maximal magnetic field in the average
time_samples = np.arange(-200, 800 + 1, 4)  # Recreate the time array
plt.axvline(x=time_samples[time_idx], color='red', linestyle='--', linewidth=2,
label='Max Field Time Point')


# Add labels and title
plt.title(f'Magnetic Field for Each Repetition (Sensor {sensor_idx + 1})')
plt.xlabel('Time (ms)')
plt.ylabel('Magnetic Field (Tesla)')
plt.grid()

# Adjust layout to make space for the legend
plt.subplots_adjust(right=0.85)  # Adjust if needed

# Show the plot
plt.show()
```



viii. Describe in your own words how the response found in the average is represented in the single repetitions. But do make sure to use the concepts *signal* and *noise* and comment on any differences on the range of values on the y-axis

The average of the magnetic field response (across repetitions of a visual stimuli) effectively shows the underlying signal while the noise is filtered out in the individual trials. The signal represents consistent patterns of a neural activity which is associated with the stimulus, this could reflect coherent brain responses that align temporally with the stimulus in the presentation. However, each repetition also contains various sources of noise, such as biological variability, measurement of errors, and unrelated brain activity, leading to variations in the recorded magnetic fields.

As a result, individual repetitions display a wider range of y-axis values due to both strong signals and significant noise. This shows peaks and lows that may not align with the average trend of the field.

Range of values on the Y-axis

The average response typically shows a narrower raneg of values compared to indiviual repetitions, because aveaging tends to limit the extreme variations, leading to a more stable representation of the signal. The individual repetitions however, show a wieder spread in the y-axis values, which might be due to the presence of both strong signals and high leevl of noise. Meaning that some repetitions might both show sharp peaks and lows which doesn't align with the average trend.

Alex

2) Now load `pas_vector.npy` (call it `y`). PAS is the same as in Assignment 2, describing the clarity of the subjective experience the subject reported after seeing the briefly presented stimulus

```
file_path = os.path.expanduser('~/Documents/CogSci/Sem3/Methods3/pas_vector.npy')
y = np.load(file_path) # Loading the file path to the variable
```

i. Which dimension in the ‘data‘ array does it have the same length as?

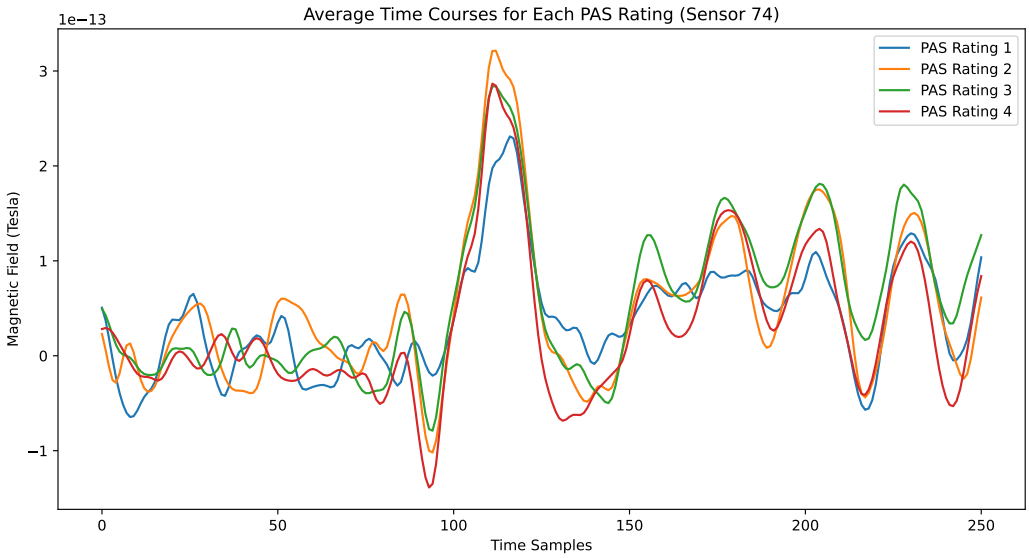
```
y = np.squeeze(y)
target_length = len(y)
print(target_length)
```

```
## 682
```

```
# The dimension of which it shares the same length is the number of repetitions.
```

ii. Now make four averages (As in Exercise A.1.iv), one for each PAS rating, and plot the four time courses (one for each PAS rating) for the sensor found in Exercise A.1.vi

```
# Creating a variable to store average time courses for each PAS rating
average_time_courses = {}

# Loop through each PAS rating (assuming ratings are 1, 2, 3, and 4)
for rating in range(1, 5):
    # Getting the indices of all occurrences of the PAS rating in y
    indices = np.where(y == rating)[0]

    # Select data for the chosen PAS rating across all repetitions and sensors
    selected_data = data[indices, :, :]

    # Choose a specific sensor index, e.g., sensor 0
    sensor_data = selected_data[:, 73, :]  # Selecting the time courses for sensor 73

    # Calculate the average time course for this sensor and PAS rating
    average_time_course = np.mean(sensor_data, axis=0)

    # Store the average time course in the dictionary
    average_time_courses[rating] = average_time_course

# Plotting the average time courses for each PAS rating
plt.figure(figsize=(12, 6))
for rating, time_course in average_time_courses.items():
    plt.plot(time_course, label=f'PAS Rating {rating}')

# Adding labels and legend
plt.title("Average Time Courses for Each PAS Rating (Sensor 74)")
plt.xlabel("Time Samples")
plt.ylabel("Magnetic Field (Tesla)")
plt.legend()
plt.show()
```



iii. Notice that there are two early peaks (measuring visual activity from the brain), one before 200 ms and one around 250 ms. Describe how the amplitudes of responses are related to the four PAS-scores. Does PAS 2 behave differently than expected?

Initial Peak (before 200 ms): At the first peak, the amplitudes vary slightly across the PAS scores. Notably, PAS 2 shows an unexpectedly high amplitude compared to PAS 1 and even PAS 3. This is surprising, as one would typically expect PAS 2 to have a lower amplitude than PAS 3 if the ratings are intended to show increasing strength of subjective experience.

Second Peak (around 250 ms): In the second peak, amplitude differences across PAS scores are again visible. PAS 4 displays the highest response amplitude, which aligns with its rating as the strongest subjective experience. However, PAS 2 again shows an amplitude similar to or slightly higher than PAS 1 and PAS 3, which diverges from the expected pattern.

PAS 2 behaves differently than expected, as it shows amplitudes similar to or greater than PAS 3 and even occasionally closer to PAS 4.

EXERCISE B - Do logistic regression to classify pairs of PAS-ratings

Emma

1) Now, we are going to do Logistic Regression with the aim of classifying the PAS-rating given by the subject

    i. We'll start with a binary problem - create a new array called `data_1_2` that only contains PAS responses 1 and 2. Similarly, create a `y_1_2` for the target vector

```
pas_1_2_indices = np.where((y == 1) | (y == 2))
y_1_2 = y[pas_1_2_indices[0]]
data_1_2 = data[pas_1_2_indices[0]]
```

ii. Scikit-learn expects our observations (‘data_1_2‘) to be in a 2d-array, which has samples (repetitions) on dimension 1 and features (predictor variables) on dimension 2.

Our ‘data_1_2‘ is a three-dimensional array. Our strategy will be to collapse our two last dimensions (sensors and time) into one dimension, while keeping the first dimension as it is (repetitions). Use ‘np.reshape‘ to create a variable ‘X_1_2‘ that fulfils these criteria.

```
X_1_2 = np.reshape(data_1_2, (data_1_2.shape[0],(num_sensors*num_time_samples)))
```

iii. Import the ‘StandardScaler‘ and scale ‘X_1_2‘

```
X_1_2_z = StandardScaler().fit_transform(X_1_2)
```

iv. Do a standard ‘LogisticRegression‘ - can be imported from ‘sklearn.linear_model‘ - make sure there is no ‘penalty‘ applied

```
def logistic_regression(X_train, X_test, y_train, y_test, solver, penalty):
    clf = LogisticRegression(penalty = penalty, solver = solver, random_state = 1)
    clf.fit(X_train, y_train)
    scoring = clf.score(X_test,y_test)
    coef = clf.coef_[0]
    track_nonzero_coef = np.sum(coef != 0)
    return(coef, scoring, track_nonzero_coef)
```

```
X_1_2_train, X_1_2_test, y_1_2_train, y_1_2_test = train_test_split(X_1_2, y_1_2,
test_size = 0.3, random_state = 1)
scaling = StandardScaler()
X_1_2_train_z =  scaling.fit_transform(X_1_2_train)
X_1_2_test_z = scaling.transform(X_1_2_test)
```

v. Use the ‘score‘ method of ‘LogisticRegression‘ to find out how many labels were classified correctly. Are we overfitting? Besides the score, what would make you suspect that we are overfitting?

```
coef_none,acc_none, track_nonzero_coef_none = logistic_regression(X_1_2_train_z,
X_1_2_test_z, y_1_2_train, y_1_2_test, penalty = None, solver = 'saga')

print('Accuracy of the classifier without using any penalty: ', acc_none)
```

```
## Accuracy of the classifier without using any penalty:  0.5384615384615384
```

```
print('Number of non-zero coefficients of the classifier without using any penalty: ',
track_nonzero_coef_none)
```

```
## Number of non-zero coefficients of the classifier without using any penalty:  25602
```

```
# We are definitely overfitting, not only can we tell by the accuracy but also from the
# fact that each of the sensors has a coefficient (no 0 coefficients).
```

vi. Now apply the _L1_ penalty instead, using the default C, - how many of the coefficients (‘.coef_‘) are non-zero after this?

```
coef_l1,acc_l1, track_nonzero_coef_l1 = logistic_regression(X_1_2_train_z, X_1_2_test_z,
y_1_2_train, y_1_2_test, penalty = 'l1', solver = 'liblinear')

print('Accuracy of the classifier with using l1 penalty: ', acc_l1)
```

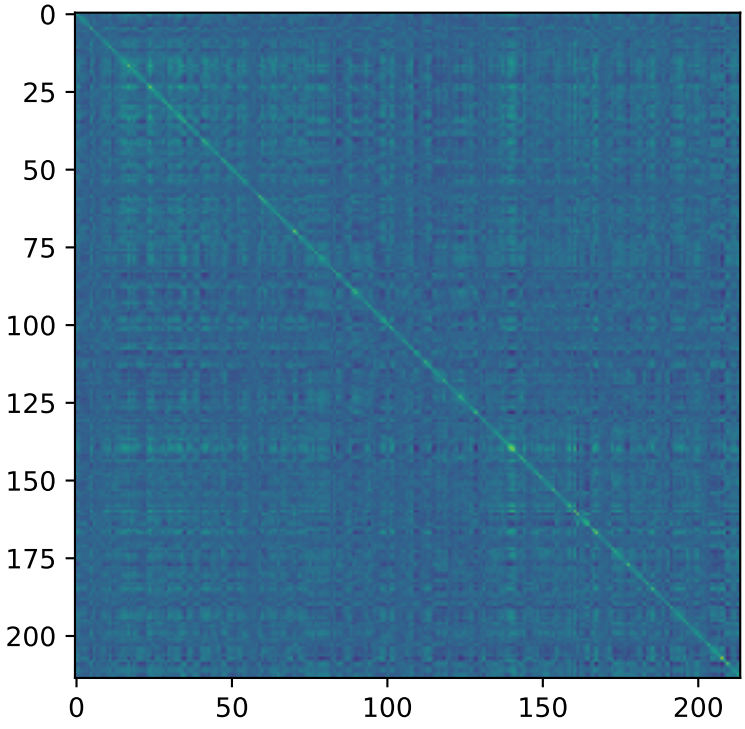
```
## Accuracy of the classifier with using l1 penalty:  0.6
```

```
print('Number of non-zero coefficients of the classifier with using l1 penalty: ',
track_nonzero_coef_l1)
```

```
## Number of non-zero coefficients of the classifier with using l1 penalty:  198
```

vii. Create a new reduced X that only includes the non-zero coefficients - show the covariance of the non-zero features; $X_{reduced}X_{reduced}^T$. Plot the covariance of the features using ‘plt.imshow‘. Compared to a covariance-plot where you do not use any penalties, do you see less covariance? (Ignore the absolute values, but look at the pattern)

```
def covar_matrix2(coef):
    covar = coef @ coef.T
    return(covar)

coef_nonzero_l1 = np.where(coef_l1 != 0)
X_1_2_reduced = np.array(X_1_2[:,coef_nonzero_l1[0]])
plt.imshow(covar_matrix2(X_1_2_reduced))
# There is less covariance within the reduced feature vector matrix than in the
# non-reduced one.
```



2) Now, we are going to build better (more predictive) models by using cross-validation as an outcome measure

   i. Import `cross_val_score` and `StratifiedKFold` from `sklearn.model_selection`

```
# Imported at the start
```

ii. To make sure that our training data sets are not biased to one target (PAS) or the other, create ‘y_1_2_equal‘, which should have an equal number of each target. Create a similar ‘X_1_2_equal‘. The function ‘equalize_targets_binary‘ in the code chunk associated with Exercise B.2.ii can be used. Remember to scale ‘X_1_2_equal‘!

```
def equalize_targets_binary(data, y):
    np.random.seed(7)
    targets = np.unique(y) ## find the number of targets
    if len(targets) > 2:
        raise NameError("can't have more than two targets")
    counts = list()
    indices = list()
    for target in targets:
        counts.append(np.sum(y == target)) ## find the number of each target
        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=False)
```

```
    second_choice = np.random.choice(indices[1], size=min_count,replace=False)

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

    return new_data, new_y

data_1_2_eq, y_1_2_eq = equalize_targets_binary(data_1_2, y_1_2)
```

iii. Do cross-validation with 5 stratified folds doing standard ‘LogisticRegression‘ (See Exercise B.1.iv)

```
X_1_2_eq = np.reshape(data_1_2_eq, (data_1_2_eq.shape[0],(num_sensors*num_time_samples)))
X_1_2_eq_z = scaling.transform(X_1_2_eq)
```

iv. Do L2-regularisation with the following ‘Cs=[1e5, 1e1, 1e-5]‘. Use the same kind of cross-validation as in Exercise B.2.iii. In the best-scoring of these models, how many more/fewer predictions are correct (on average)?

```
def cross_val_classifiers(X, y, C):
    clf = LogisticRegression(penalty = 'l2', solver = 'liblinear', random_state=1, C = C)
    clf.fit(X,y)
    scoring_cross_val_mn = np.mean(cross_val_score(clf, X, y , cv=5))
    print('Mean accuracy of classifier with C = ', C , ':',np.mean(scoring_cross_val_mn))

cross_val_classifiers(X_1_2_eq_z, y_1_2_eq, C = 1e5)
```

```
## Mean accuracy of classifier with C =  100000.0 : 0.5147435897435897
```

```
cross_val_classifiers(X_1_2_eq_z, y_1_2_eq, C = 1e1)
```

```
## Mean accuracy of classifier with C =  10.0 : 0.525
```

```
cross_val_classifiers(X_1_2_eq_z, y_1_2_eq, C = 1e-5)
```

```
## Mean accuracy of classifier with C =  1e-05 : 0.560128205128205
```

```
# On average approx 6% more of the samples are predicted correctly than in the other ones.
# That equals to about 11 more correct predictions
```

v. Instead of fitting a model on all ‘n_sensors * n_samples‘ features, fit a logistic regression (same kind as in Exercise B.2.iv (use the ‘C‘ that resulted in the best prediction)) for __each__ time sample and use the same cross-validation as in Exercise B.2.iii. What are the time points where classification is best? Make a plot with time on the x-axis and classification score on the y-axis with a horizontal line at the chance level (what is the chance level for this analysis?)

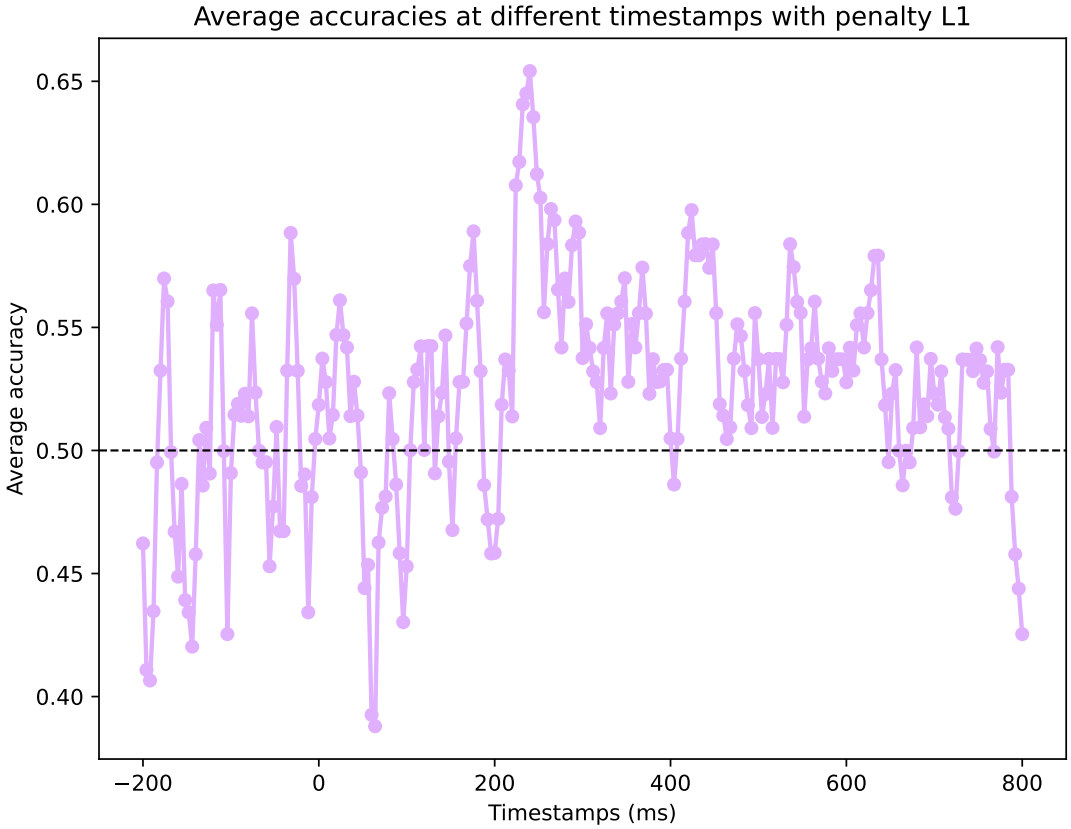
```
# Reshape the data to a flat array for timestamps
X_1_2_timestamps = data_1_2.reshape(-1)  # Flatten into 1D array

# Reshape to (reps * sensors, time_samples) for easier scaling
X_reshaped = data_1_2.reshape(-1, data_1_2.shape[2])
X_reshaped_z = scaling.fit_transform(X_reshaped)  # Apply scaling
X_1_2_timestamps_z = X_reshaped_z.reshape(data_1_2.shape)  # Reshape back

# Function to fit a logistic regression model on each time sample
def fit_each_timesample(data, y, penalty, C):
    avg_acc = []
    reps, sensors, time_samples = data.shape
    clf = LogisticRegression(penalty=penalty, solver='liblinear', random_state=1, C=C)

    for i in range(time_samples):
        X = data[:, :, i].reshape(reps, sensors)  # Creates matrix for this time sample
        scoring_cross_val = cross_val_score(clf, X, y, cv=5)
        avg_acc.append(np.mean(scoring_cross_val))

    return avg_acc

# Run the logistic regression for each time sample with specified penalties and C values
accuracies_l1 = fit_each_timesample(X_1_2_timestamps_z, y_1_2, penalty='l1', C=1e-1)
accuracies_l2 = fit_each_timesample(X_1_2_timestamps_z, y_1_2, penalty='l2', C=1e-5)

# Identify peak performance time point
best_time_point = np.argmax(accuracies_l1)
print("Peak performance was at time", time_samples[best_time_point], "ms.")
```

```
## Peak performance was at time 240 ms.
```

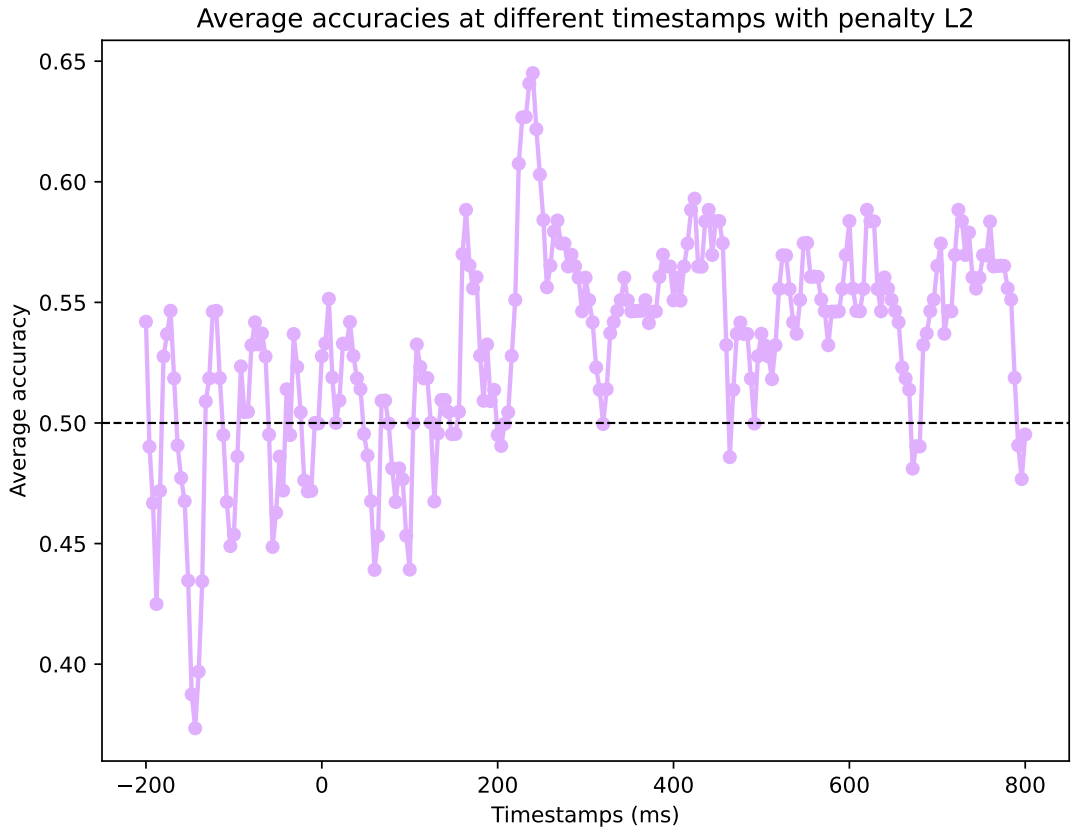
```
# Define chance level (assuming binary classification, it's 0.5)
chance_level = 0.5

# Plotting function with time on x-axis and accuracy on y-axis
def plot_accuracy_timestamps(time_samples, accuracies, penalty_desc):
    plt.figure(figsize=(8, 6))
    plt.plot(time_samples, accuracies, color="#E0B0FF", linestyle="-", linewidth=2)
    plt.axhline(y=chance_level, color="black", linestyle="--", linewidth=1)
    plt.scatter(time_samples, accuracies, color="#E0B0FF", marker="o", s=30)
    plt.title("Average accuracies at different timestamps " + penalty_desc)
    plt.xlabel("Timestamps (ms)")
    plt.ylabel("Average accuracy")
    plt.show()
```

vi. Now do the same, but with L1 regression - set ‘C=1e-1‘ - what are the time points when classification is best? (make a plot)?

```
plot_accuracy_timestamps(times, accuracies_l1, penalty_desc = 'with penalty L1')
```

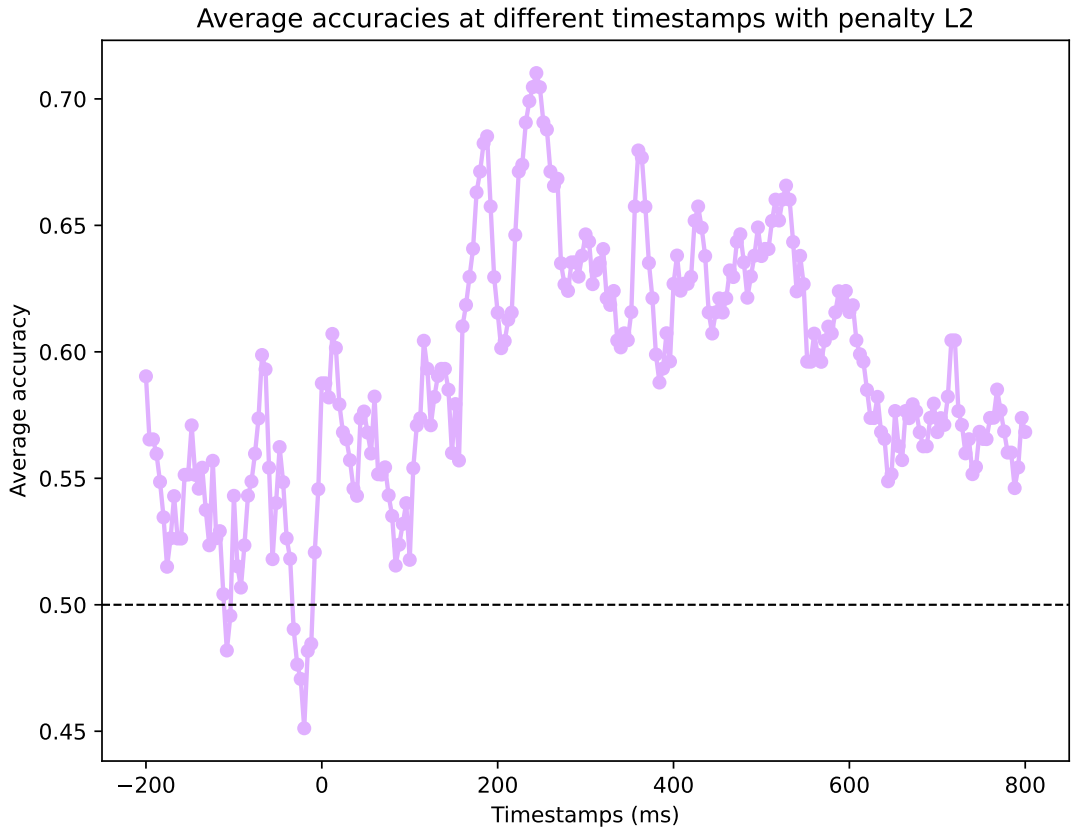



```
plot_accuracy_timestamps(times, accuracies_l2, penalty_desc = 'with penalty L2')
```



vii. Finally, fit the same models as in Exercise B.2.vi but now for `data_1_4` and `y_1_4` (create a data set and a target vector that only contains PAS responses 1 and 4). What are the time points when classification is best? Make a plot with time on the x-axis and classification score on the y-axis with a horizontal line at the chance level (what is the chance level for this analysis?)

```
pas_1_4_indices = np.where((y == 1) | (y == 4))
y_1_4 = y[pas_1_4_indices[0]]
data_1_4 = data[pas_1_4_indices[0]]
X_1_4 = np.reshape(data_1_4, (data_1_4.shape[0],(num_sensors*num_time_samples)))

X_reshaped2 = data_1_4.reshape(-1, data_1_4.shape[2])
X_reshaped2_z = scaling.fit_transform(X_reshaped2)
X_1_4_timestamps_z = X_reshaped2_z.reshape(data_1_4.shape)

accuracies_l2_1_4 = fit_each_timesample(X_1_4_timestamps_z, y_1_4, penalty = 'l2',
C = 1e-5)

plot_accuracy_timestamps(times, accuracies_l2_1_4, penalty_desc = 'with penalty L2')
```



viii. Is pairwise classification of subjective experience possible? Any surprises in the classification accuracies, i.e. how does the classification score for PAS 1 vs 4 compare to the classification score for PAS 1 vs 2?

Pairwise classification is possible when the foundation is based on sensor signals from specific timesteps. The performance peaks at around the sensor inputs from 240 ms after onset. The performance for classifying between PAS 1 and 4 is always over 0.7 accuracy vs when classifying between PAS 1 and 2 it is never over 0.7 accuracy.

Alex

3) Do multinomial logistic regression

   i. First equalize the number of targets using the function associated with each PAS-rating using the function associated with Exercise B.3.i

```
# Exercise B.3.i
def equalize_targets(data, y):
    np.random.seed(7) # Setting a random seed
    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) # Checking the minimum number of counts per PAS rating
```

```
    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, # Collecting the indices
                                  third_choice, fourth_choice))
    new_y = y[new_indices]
    new_data = data[new_indices, :, :]

    return new_data, new_y

equalized_data, equalized_y = equalize_targets(data,y)  # Creating new variables using
# the associated function to equalize.
```

ii. Split the equalized data set (with all four ratings) into a training part and test part, where the test part is 30 % of the trials. Use ‘train_test_split‘ from ‘sklearn.model_selection‘. Also standardise the data.

```
def flatten_data(X): # A function to reshape the 3d array into a 2d so that we can
  # transform it.
    return X.reshape(X.shape[0], -1)

X_train, X_test, y_train, y_test = train_test_split( # Creating training and test
  # variables for each.
    equalized_data, equalized_y, test_size=0.3, random_state=1) # Using 30% of the data
    # for tests.

pipe_n1 = Pipeline([
    ('reshape', FunctionTransformer(flatten_data)),  # Flatten the data to 2D model
    # for standardization
    ('scl', StandardScaler()),  # Applying standardization
])

# Fitting the pipeline on training data
pipe_n1.fit(X_train, y_train)
```

```
## Pipeline(steps=[('reshape',
##                  FunctionTransformer(func=<function flatten_data at 0x1783918a0>)),
##                 ('scl', StandardScaler())])
```

iii. Use the L2-regularisation chosen in Exercise B.2.iv (use the ‘C‘ that resulted in the best prediction) ‘fit‘the training set and ‘predict‘ on the test set. This time your features are the number of sensors multiplied by the number of samples.

```
best_C = 1e1

# Flatten the data function
def flatten_data(X):
    return X.reshape(X.shape[0], -1)  # Reshape to [n_samples, n_features]

# Splitting the data into training and test.
```

```
X_train, X_test, y_train, y_test = train_test_split(
    equalized_data, equalized_y, test_size=0.3, random_state=1)

# Define the pipeline with L2-regularized Logistic Regression using the best predicted C
pipe_n2 = Pipeline([
    ('reshape', FunctionTransformer(flatten_data)), # Flatten to 2D
    ('scl', StandardScaler()), # Standardize features
    ('pca', PCA(n_components=2)), # Principle Component Analysis with 2 components
    ('clf', LogisticRegression(C=best_C, penalty='l2', random_state=1))# L2 regularization
])

# Fit the model on training data and predict on test data
pipe_n2.fit(X_train, y_train)
```

```
## Pipeline(steps=[('reshape',
##                  FunctionTransformer(func=<function flatten_data at 0x178391da0>)),
##                 ('scl', StandardScaler()), ('pca', PCA(n_components=2)),
##                 ('clf', LogisticRegression(C=10.0, random_state=1))])
```

```
print('Training Accuracy: %.3f' % pipe_n2.score(X_train, y_train))
```

```
## Training Accuracy: 0.314
```

```
print('Test Accuracy: %.3f' % pipe_n2.score(X_test, y_test))
```

```
## Test Accuracy: 0.328
```

iv. Create a _confusion matrix_. It is a 4x4 matrix. The row names and the column names are the PAS-scores. There will thus be 16 entries. The PAS1xPAS1 entry will be the number of actual PAS1, y_{pas1} that were predicted as PAS1, $\hat y_{pas1}$. The PAS1xPAS2 entry will be the number of actual PAS1, y_{pas1} that were predicted as PAS2, $\hat y_{pas2}$ and so on for the remaining 14 entries. Plot the matrix

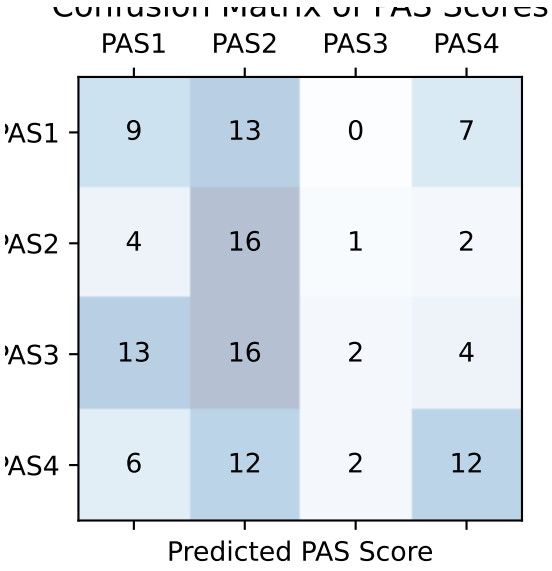
```
# Fit the model on training data and predict on test data again
pipe_n2.fit(X_train, y_train)
```

```
## Pipeline(steps=[('reshape',
##                  FunctionTransformer(func=<function flatten_data at 0x178391da0>)),
##                 ('scl', StandardScaler()), ('pca', PCA(n_components=2)),
##                 ('clf', LogisticRegression(C=10.0, random_state=1))])
```

```
y_pred = pipe_n2.predict(X_test)
# Creating the confusion matrix from the predicted data and test data.
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)
```

```
## [[ 9 13  0  7]
##  [ 4 16  1  2]
##  [13 16  2  4]
##  [ 6 12  2 12]]
```

```
# Creating custom labels for the confusion matrix
pas_labels = ['PAS1', 'PAS2', 'PAS3', 'PAS4']
# Plotting a better looking confusion matrix.
fig, ax = plt.subplots(figsize=(3, 3))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
  for j in range(confmat.shape[1]):
    ax.text(x=j, y=i,
            s=confmat[i, j],
            va='center', ha='center')

# Adding the custom labels to the axis.
ax.set_xticks(range(len(pas_labels)))
ax.set_xticklabels(pas_labels)
ax.set_yticks(range(len(pas_labels)))
ax.set_yticklabels(pas_labels)
# Adding heading labels on the plot and both axis.
plt.xlabel('Predicted PAS Score')
plt.ylabel('Actual PAS Score')
plt.title('Confusion Matrix of PAS Scores')
plt.show()
```



v. Based on the confusion matrix, describe how ratings are misclassified and if that makes sense given that ratings should measure the strength/quality of the subjective experience. Is the classifier biased towards specific ratings?

The classifier is bias towards a weaker rating in particular PAS2 as it under-predicts the PAS3 and 4 ratings whilst predicting the majority of experiences as a PAS2 rating. This makes sense as it is subjective experience and could be explained by an inherent overlap in the distinguishing parameters used to identify each level of experience, which would explain the difficulty the model had in identifying differences in level of subjective experience.

```
# Exercise B.2.ii
def equalize_targets_binary(data, y):
    np.random.seed(7)
    targets = np.unique(y) ## find the number of targets
    if len(targets) > 2:
        raise NameError("can't have more than two targets")
    counts = list()
    indices = list()
    for target in targets:
        counts.append(np.sum(y == target)) ## find the number of each target
        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=False)
    second_choice = np.random.choice(indices[1], size=min_count,replace=False)

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

    return new_data, new_y
# Exercise B.3.i

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
```