practical_exercise_8 , Methods 3, 2021, autumn semester

Anja, Jessica, Magnus, Juli, Astrid

17/11/2021

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

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 3 and will be part of your final portfolio

EXERCISE 1 - 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)

##1.1) 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

```
library(reticulate)
options(reticulate.repl.quiet = TRUE)
library(Rcpp)
```

```
import numpy as np
import matplotlib.pyplot as plt
data = np.load("data/megmag_data.npy")
```

###1.1.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?

```
data.shape
```

```
## (682, 102, 251)
```

X - number of repetitions Y - number of sensors Z - time

###1.1.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.

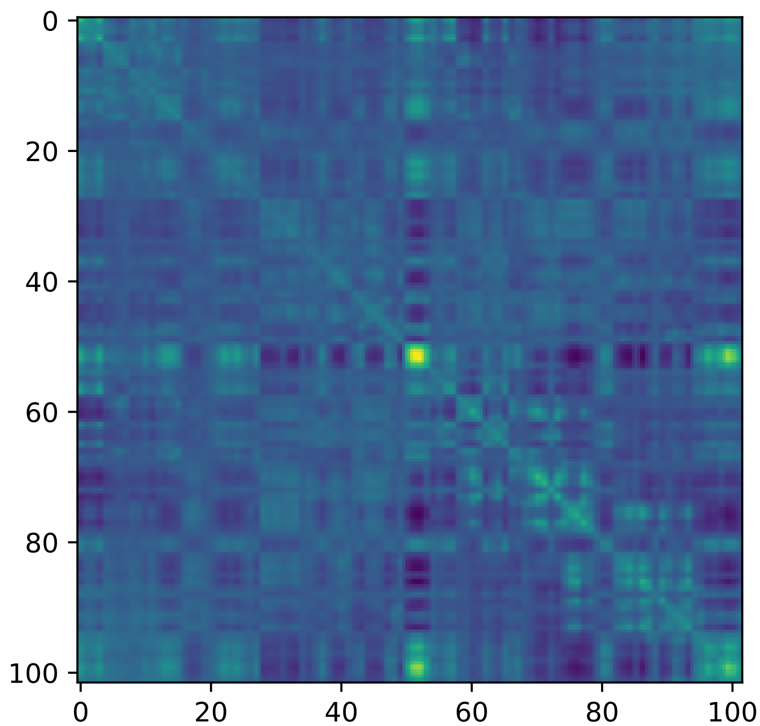
```
times = np.arange(-200, 804, 4) # just needs to be higher than top range
```

###1.1.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 = 682
covariance = []

for i in range(n):
  covariance.append(data[i,:,:] @ data[i,:,:].T)


covariance = sum(covariance)/n

plt.figure()
plt.imshow(covariance)
plt.show()
```




```
plt.cla()
```

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

```
# axis= 0 means columns, axis = 1 would be to look at the rows
rep_mean = np.mean(data, axis=0)
rep_mean = rep_mean.T
rep_mean.shape
```

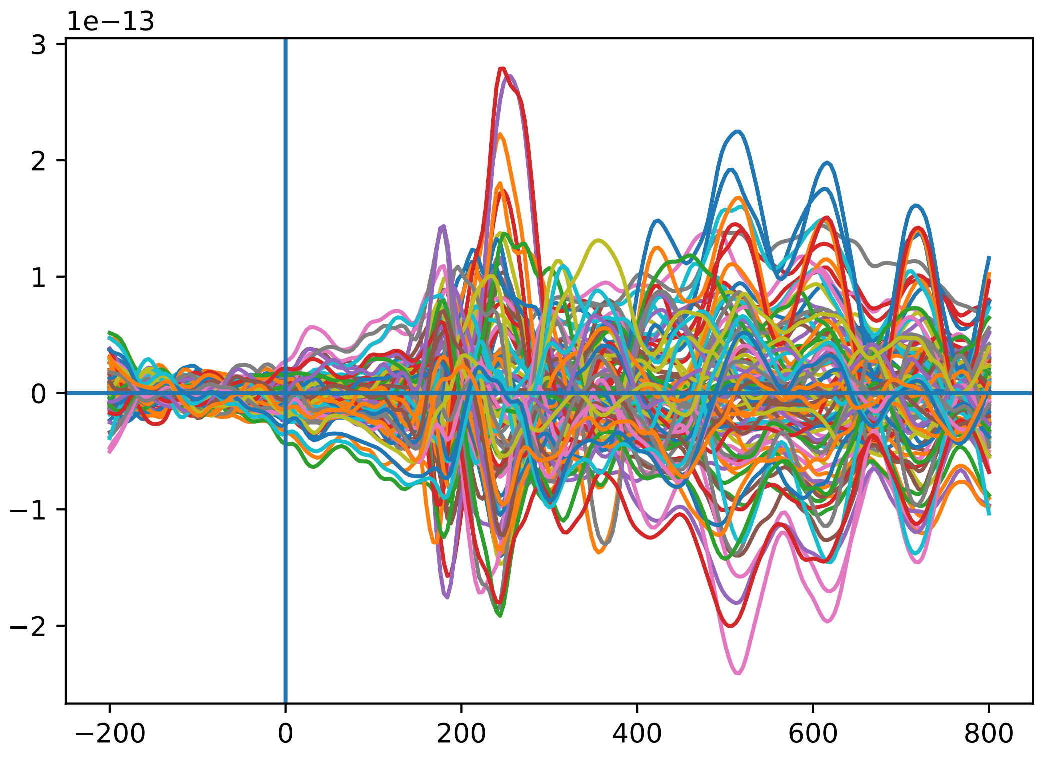
```
## (251, 102)
```

```
len(rep_mean)
```

```
## 251
```

###1.1.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`

```
plt.figure()
plt.plot(times, rep_mean)
plt.axvline(0)
plt.axhline(0)
plt.show()
```



###1.1.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.

```
np.amax(rep_mean)
```

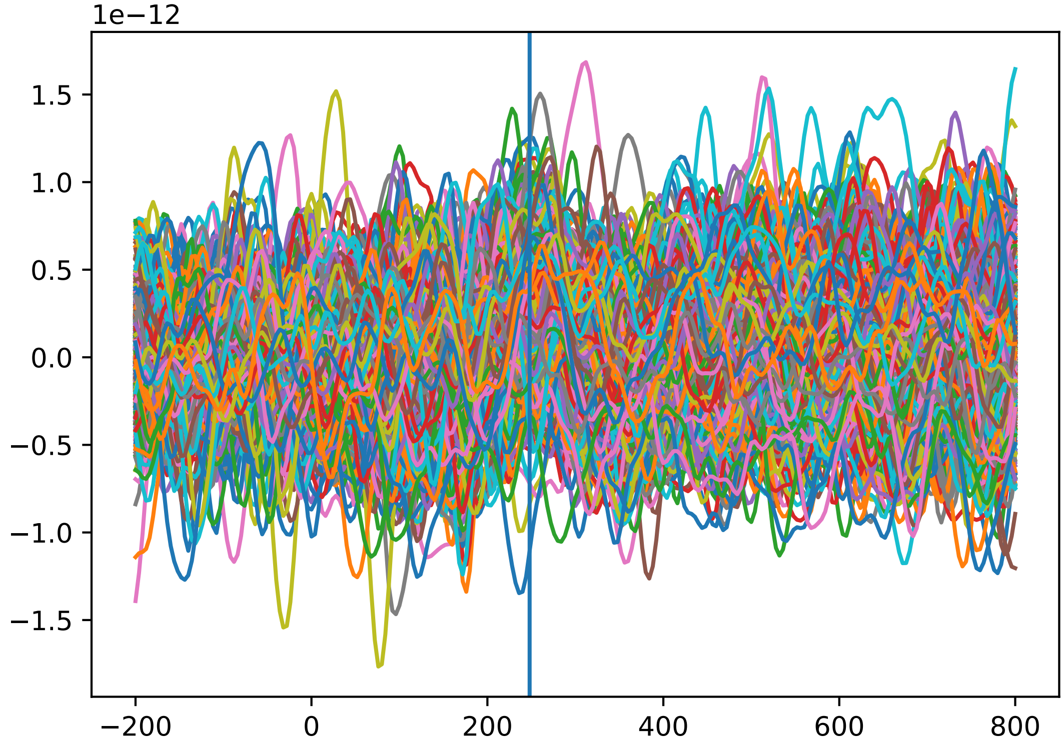
```
## 2.7886216843591933e-13
```

```
np.unravel_index(np.argmax(rep_mean), rep_mean.shape)
```

```
## (112, 73)
```

###1.1.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 1.1.v) using `plt.axvline`

```python
# plt.figure()
# plt.plot(times, data[:,73])
# plt.axvline(112)
# plt.show()

# switched 251 to 682 b/c thats the number of repetitions
for i in range(682):
    plt.plot(times, data[i,73, :])

plt.axvline(times[112])  # have to specify where it's coming from. It's 112 in the times
plt.show()
```



###1.1.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 - This plot shows all 682 repetitions at sensor 74 (indexed as 73). - We see this average in plot 1.1.v as the highest peak (red line). - At this line though, it appears like there are less negative values and more positive values. - This plot is less clear. It seems there is a lot more noise in this plot compared to plot 1.1.v where the averages are taken. There we see the signal more clearly.

##1.2) 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

```
y = np.load("data/pas_vector.npy")
```

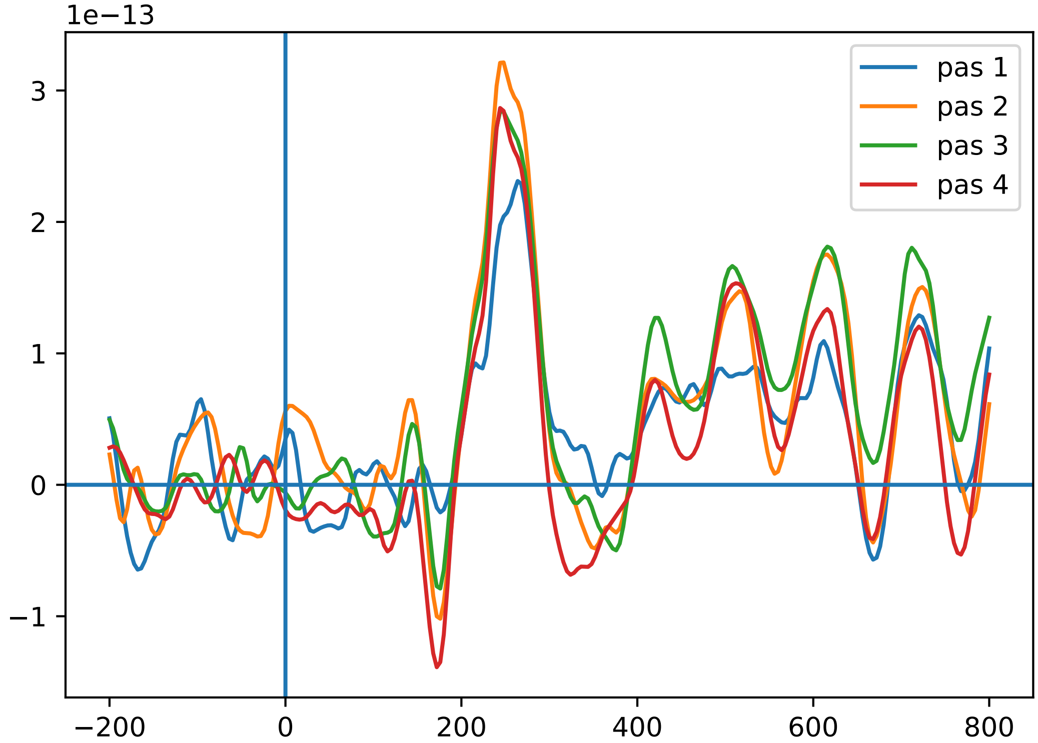
###1.2.i. Which dimension in the `data` array does it have the same length as?

```
y.shape
```

```
## (682,)
```

###1.2.ii. Now make four averages (As in Exercise 1.1.iii), one for each PAS rating, and plot the four time courses (one for each PAS rating) for the sensor found in Exercise 1.1.v

```
pas1 = np.where(y == 1)
pas2 = np.where(y == 2)
pas3 = np.where(y == 3)
pas4 = np.where(y == 4)



sensor73 = data[:,73,:]
avgpas1 = np.mean(sensor73[pas1], axis = 0)
avgpas2 = np.mean(sensor73[pas2], axis = 0)
avgpas3 = np.mean(sensor73[pas3], axis = 0)
avgpas4 = np.mean(sensor73[pas4], axis = 0)
plt.figure()
plt.plot(times, avgpas1)
plt.plot(times, avgpas2)
plt.plot(times, avgpas3)
plt.plot(times, avgpas4)
plt.axvline()
plt.axhline()
plt.legend(['pas 1', 'pas 2', 'pas 3', 'pas 4'])
plt.show()
#the labels seem random, but I double-checked them and they are correct:)
```



###1.2.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? - We expect because pas 2 and 3 to be the most chosen ratings because the are more in the middle, pas 1 and 4 are the more extreme choices. - No pas 2 behaves like we would expect. In the first 200-250 ms, people are rating pas 2 the most but as they have more time pas 3 becomes the most used.

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

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

###2.1.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

```
pas12 = np.where((y == 1) | (y == 2))

data_1_2 = np.squeeze(data[pas12,:,:]) # np.squeeze gets rid of the point that is only 1

data_1_2.shape
```

```
## (214, 102, 251)
```

```
data_1_2.ndim # how many dimensions
```

```
## 3
```

```
y_1_2 = np.squeeze(y[pas12])
```

```
len(y_1_2)
```

```
## 214
```

###2.1.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 = data_1_2.reshape(214, -1) # do we need data here or np.reshape, represents length of trials

X_1_2.ndim
```

```
## 2
```

```
X_1_2.shape

# -1, leaves the first dimension as it is, and 2 to collapse from 3 dimensions

# https://stackoverflow.com/questions/18757742/how-to-flatten-only-some-dimensions-of-a-numpy-array
```

```
## (214, 25602)
```

###2.1.iii. Import the `StandardScaler` and scale `X_1_2`

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler() # need this for it to work
X_1_2 = scaler.fit_transform(X_1_2)
```

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

```
# import logisticregression and naming it
from sklearn.linear_model import LogisticRegression
regressor = LogisticRegression(penalty = 'none') # solver default is lbfgs
log_fit = regressor.fit(X_1_2, y_1_2)

log_fit.coef_

# penalty = L1 (lambda 1) penalizing for large coefficients, solver = 'liblinear' works with L1 and L2
```

```
## array([[-0.01276554, -0.01750771, -0.02204876, ..., -0.0123659 ,
##         -0.01064456, -0.0077446 ]])
```

###2.1.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?

```
log_fit.score(X_1_2, y_1_2)
```

```
## 1.0
```

Yes we're overfitting based on the score of 1.0. Our model also fits our whole data set which often leads to overfitting. This is why it's better to split up the data into a training and test set.

###2.1.vi. Now apply the *L1* penalty instead - how many of the coefficients (`.coef_`) are non-zero after this?

```
from sklearn.linear_model import LogisticRegression
np.random.seed(7)
```

```
regressor = LogisticRegression(penalty = 'l1', solver = 'liblinear') # solver default is lbfgs
log_fit_1 = regressor.fit(X_1_2, y_1_2)

log_fit_1.coef_
```

```
## array([[0., 0., 0., ..., 0., 0., 0.]])
```

```
np.count_nonzero(log_fit_1.coef_) #counting the non-zero coefficients
```

```
## 271
```

###2.1.vii. Create a new reduced X that only includes the non-zero coefficients - show the covariance of the non-zero features (two covariance matrices can be made; $X_{reduced}X_{reduced}^T$ or $X_{reduced}^T X_{reduced}$ (you choose the right one)) . Plot the covariance of the features using `plt.imshow`. Compared to the plot from 1.1.iii, do we see less covariance?

```
non_zero = np.where(log_fit_1.coef_ != 0)
print(non_zero) # this gives us the second column which has the non zero coefficients only
```

```
## (array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
##        0, 0, 0, 0, 0, 0, 0]), array([  508,   509,  1073,  1074,  1075,  1147,  1148,  1166,  1167,
##         1188,  1189,  1194,  1241,  1242,  1243,  1508,  1509,  1539,
##         1758,  1759,  1879,  1900,  2101,  2327,  2328,  2577,  2578,
##         2594,  2595,  2616,  2617,  2978,  3015,  3016,  3023,  3024,
##         3040,  3041,  3042,  3174,  3175,  3685,  3767,  3768,  3789,
##         3819,  3820,  3821,  4110,  4344,  4345,  4356,  4430,  4544,
##         4733,  4890,  4937,  5018,  5019,  5294,  5362,  5363,  5384,
##         6034,  6035,  6115,  6116,  6291,  6343,  6389,  6531,  6540,
##         6541,  6634,  6635,  6721,  6722,  6723,  6759,  6760,  6781,
##         6782,  6870,  6871,  7075,  7144,  7145,  7621,  7622,  7792,
##         7853,  7889,  8318,  8641,  8642,  8643,  8644,  8645,  8646,
##         8647,  8727,  8728,  8802,  9321,  9322,  9568,  9606,  9714,
##         9715,  9726,  9727,  9728,  9729,  9835,  9836, 10076, 10084,
##        10520, 10532, 10533, 11320, 11555, 11556, 11687, 11702, 11703,
##        11797, 11807, 11860, 11870, 11871, 11872, 11938, 11939, 12322,
##        12367, 12368, 12369, 13074, 13075, 13666, 14001, 14002, 14375,
##        14376, 14377, 15066, 15067, 15102, 15103, 15165, 15244, 15245,
##        15276, 15561, 15923, 15924, 15925, 16026, 16277, 16315, 16485,
##        16486, 16617, 16629, 16630, 16668, 16712, 16713, 16834, 16860,
##        16867, 17231, 17232, 17295, 17511, 17885, 17886, 18098, 18117,
##        18118, 18119, 18120, 18220, 18221, 18228, 18233, 18240, 18306,
##        18307, 18330, 18331, 18389, 18573, 18601, 18632, 18895, 18896,
##        18948, 19147, 19148, 19161, 19162, 19199, 19246, 19247, 19353,
##        19382, 19384, 19385, 19636, 19676, 19677, 19749, 19887, 19888,
##        19896, 19897, 19937, 19938, 19939, 19959, 19960, 20451, 20452,
```

```
##        20707, 20708, 20882, 20883, 21342, 21343, 21344, 21496, 21497,
##        21527, 21528, 21671, 21763, 21764, 21976, 21984, 21985, 21986,
##        22094, 22095, 22130, 22181, 22182, 22183, 22196, 22197, 22437,
##        22438, 22522, 22523, 22524, 22611, 22612, 22688, 22689, 22690,
##        23600, 23601, 23901, 23902, 23963, 24360, 25213, 25214, 25451,
##        25452]))
```

```
print(non_zero[1])
```

```
## [  508   509  1073  1074  1075  1147  1148  1166  1167  1188  1189  1194
##   1241  1242  1243  1508  1509  1539  1758  1759  1879  1900  2101  2327
##   2328  2577  2578  2594  2595  2616  2617  2978  3015  3016  3023  3024
##   3040  3041  3042  3174  3175  3685  3767  3768  3789  3819  3820  3821
##   4110  4344  4345  4356  4430  4544  4733  4890  4937  5018  5019  5294
##   5362  5363  5384  6034  6035  6115  6116  6291  6343  6389  6531  6540
##   6541  6634  6635  6721  6722  6723  6759  6760  6781  6782  6870  6871
##   7075  7144  7145  7621  7622  7792  7853  7889  8318  8641  8642  8643
##   8644  8645  8646  8647  8727  8728  8802  9321  9322  9568  9606  9714
##   9715  9726  9727  9728  9729  9835  9836 10076 10084 10520 10532 10533
##  11320 11555 11556 11687 11702 11703 11797 11807 11860 11870 11871 11872
##  11938 11939 12322 12367 12368 12369 13074 13075 13666 14001 14002 14375
##  14376 14377 15066 15067 15102 15103 15165 15244 15245 15276 15561 15923
##  15924 15925 16026 16277 16315 16485 16486 16617 16629 16630 16668 16712
##  16713 16834 16860 16867 17231 17232 17295 17511 17885 17886 18098 18117
##  18118 18119 18120 18220 18221 18228 18233 18240 18306 18307 18330 18331
##  18389 18573 18601 18632 18895 18896 18948 19147 19148 19161 19162 19199
##  19246 19247 19353 19382 19384 19385 19636 19676 19677 19749 19887 19888
##  19896 19897 19937 19938 19939 19959 19960 20451 20452 20707 20708 20882
##  20883 21342 21343 21344 21496 21497 21527 21528 21671 21763 21764 21976
##  21984 21985 21986 22094 22095 22130 22181 22182 22183 22196 22197 22437
##  22438 22522 22523 22524 22611 22612 22688 22689 22690 23600 23601 23901
##  23902 23963 24360 25213 25214 25451 25452]
```

```
x_reduced = X_1_2[:,non_zero[1]] # Taking the previous X_1_2 and subsetting the non-zero coefficients


print(x_reduced)
```

```
## [[-2.96649716e-01 -2.99791861e-02 -1.21738110e+00 ...  1.54952360e+00
##    1.33042866e+00  1.62291717e+00]
##  [-2.46103390e+00 -2.68268691e+00 -2.05998231e-01 ... -3.29777495e-01
##   -6.11189575e-01 -7.70767335e-01]
##  [ 3.95608023e-01  4.96007975e-01  1.09336933e+00 ...  5.00538915e-01
##   -5.70012989e-02 -1.41868725e-01]
##  ...
##  [ 1.20486074e+00  1.81774225e+00  6.01313862e-01 ...  3.62205793e-01
##   -1.62354200e-02 -8.12596149e-04]
##  [-8.00857676e-01 -8.87046622e-01 -3.51136748e-01 ... -1.26324158e+00
##   -1.49951134e+00 -1.23028692e+00]
##  [ 1.00814894e+00  1.12221470e+00 -9.73885907e-01 ... -1.10410413e+00
##   -1.31968482e+00 -1.40690227e+00]]
```

```
print(x_reduced[1])

# we do the x_reduced transposed * x_reduced so that we "get rid of" the zeros, per matrix multiplicati
```

```
## [-2.4610339  -2.68268691 -0.20599823 -0.17246033 -0.01367767 -1.40894962
##  -1.33006262 -0.49899914 -0.88005576 -0.25177442 -0.22021108  0.02350071
##   0.59522526  0.44638411  0.28965691  0.23129443 -0.05131349 -0.02623032
##   0.28983247 -0.02399064 -0.50512038 -0.9200301   0.38625745 -1.72269589
##  -1.4606957  -1.39045936 -1.34509346  0.09779979  0.3055248  -0.74513901
##  -0.89537051 -0.07788394 -0.43792754 -0.83048122  1.2213279   1.76671811
##  -0.83891532 -1.06916536 -1.31371533 -1.94580744 -2.29216896 -0.25793282
##   0.60539399  0.43622658  0.90733653 -0.87818176 -0.7979141  -0.80304948
##   0.64110772 -0.35106954 -0.18202593 -0.99918741  0.05678514  0.93699244
##   0.12596323  0.50374211  0.92273815 -0.98197429 -1.23879553 -0.38039944
##  -0.06153026 -0.43480512 -0.07807859  0.64349711  0.99470251 -0.05842724
##  -0.02064857 -0.4997193   0.25262659 -1.16580876  0.09680079 -0.72039584
##  -0.65018541  0.59617634  0.57150729 -1.5307592  -1.13901737 -0.83008878
##   0.02422272  0.04717282 -0.16959054  0.30095998  0.38507348  0.30327692
##  -0.47825598  0.15922971  0.02754739 -0.9309593  -0.63448055 -0.14223849
##   1.22771349  0.65941571  0.35825829  0.57665765  0.31517699  0.03481538
##  -0.03463139  0.23527844  0.73391564  1.17944104  1.20782213  1.23631444
##   0.66339472  0.75254299  0.7544224   0.16495118 -0.05415047 -0.04588437
##   0.07594321  0.29030813  0.60599865  0.83212402  0.84079669 -0.50137395
##  -0.63605745  0.22941748  1.16120568 -0.10877795 -0.59183066 -0.58401073
##  -1.39596579  0.82982711  0.87299106  1.29742766 -0.24806161 -0.10065245
##  -0.43933589  1.49662875 -0.2624484  -0.62713587 -0.69059774 -0.43639136
##   1.61721913  1.81407669 -1.71835031 -0.80267207 -0.5938525  -0.26588588
##   1.21022483  1.24353885 -0.40391368 -0.41412115 -0.07055581  0.11606245
##   0.06255859 -0.04419396  2.51637832  2.72578571 -1.71904473 -1.95075254
##   0.45228098 -0.47718707 -0.35915502 -0.00467857  1.43448267  0.83098645
##   0.94040797  0.98981419  0.16413738  0.43018414  1.03644961  0.44996653
##   0.49159294 -0.72952306 -0.37362983 -0.4453545  -0.49764284 -0.6785949
##  -0.7395672  -0.73292948  1.1562789  -0.12562799  1.57226926  1.93936628
##   0.20019178  1.15535793 -0.07159498 -0.30540111  0.74211307 -0.88856323
##  -0.65839809 -0.63750269 -0.76396633 -0.08611407 -0.04646693  0.02820403
##   0.80068981  0.93927029 -0.08674199  0.04795553  0.68725877  0.92178542
##   0.16987809  1.46409656  1.03563249 -0.3705145   0.45103202  0.72487515
##   0.89494485  0.53974065  0.86761005 -0.85271677 -1.24397199  0.13447246
##  -0.04470731 -0.1895388   0.29867648  0.67479342  0.10080401 -0.25086126
##   0.50273053 -0.6567863  -0.65289637  0.93345892 -0.64386918 -0.40718801
##   0.817433    1.19574108 -0.95688949 -0.884038   -0.6871083  -0.39744546
##   0.00343538  1.0288506   1.04814234 -0.09035265  0.26390983  1.64110855
##   1.08499341 -0.70070929 -0.81752453 -0.81222384  1.07138604  1.07350937
##  -0.96035386 -0.90619923 -0.65573978 -1.64541088 -1.4847335  -0.67549383
##   0.25735435  0.7054658   0.93135937 -0.32156879 -0.41760504  0.88601323
##   0.69853698  0.26196965  0.11205097 -0.78443061 -0.55902431 -0.03477922
##   0.15404829  0.01215266  0.21310537  0.32583267  0.01476294  0.17346496
##  -0.55926406 -0.34844186  0.04419791  0.41869931  0.381603    0.04070425
##   0.0944386  -1.19570824  1.68642585 -0.49739315 -0.32977749 -0.61118958
##  -0.77076734]
```

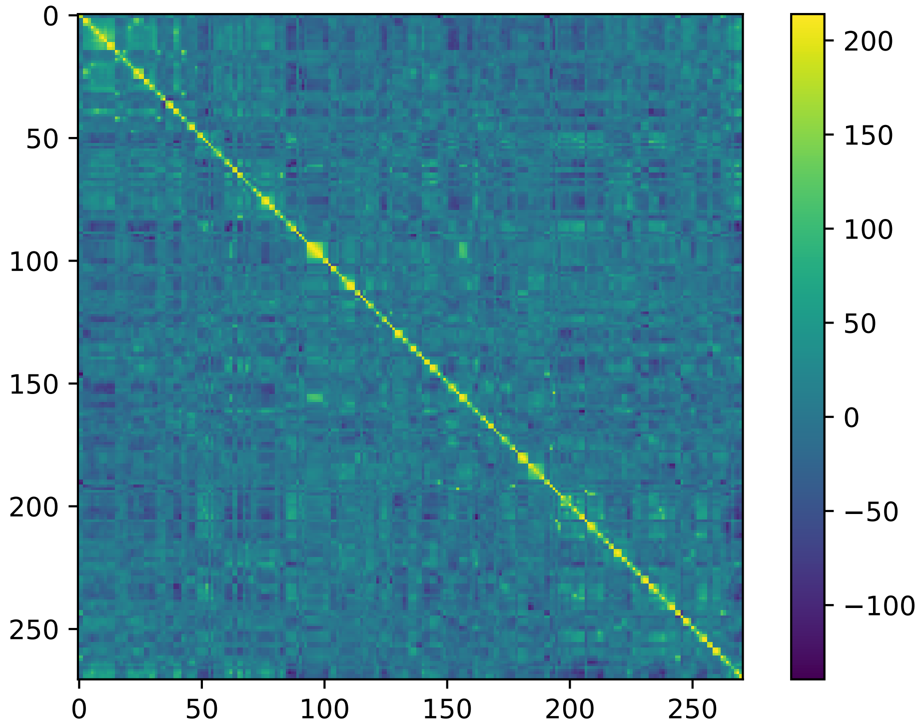
```
x_reduced_cov = (x_reduced.T @ x_reduced)
x_reduced_cov.shape
```

```
## (271, 271)
```

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

```
## <matplotlib.colorbar.Colorbar object at 0x1730c9b20>
```

```
plt.show()
```



We see more covariance in this plot ##2.2) Now, we are going to build better (more predictive) models by using cross-validation as an outcome measure
###2.2.i. Import `cross_val_score` and `StratifiedKFold` from `sklearn.model_selection`

```
from sklearn.model_selection import cross_val_score as cvs
from sklearn.model_selection import StratifiedKFold as skfold
```

###2.2.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 2.2.ii can be used. Remember to scale `X_1_2_equal`!

```
# Function is to get same number of trials with post pas 1 and pas 2 rating; takes the minimum number,
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.shape
```

```
## (214, 102, 251)
```

```
y_1_2.shape

# Use the function
```

```
## (214,)
```

```
data_1_2_equal, y_1_2_equal = equalize_targets_binary(data_1_2, y_1_2)

data_1_2_equal.shape
```

```
## (198, 102, 251)
```

```
y_1_2_equal.shape

# Reshape data into 2d
```

```
## (198,)
```

```
X_1_2_equal = data_1_2_equal.reshape(198, -1)

X_1_2_equal.shape

# Scale the data
```

```
## (198, 25602)
```

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_1_2_equal = scaler.fit_transform(X_1_2_equal)
```

###2.2.iii. Do cross-validation with 5 stratified folds doing standard `LogisticRegression` (See Exercise 2.1.iv)

```
# stratified folds
from sklearn.linear_model import LogisticRegression
regressor = LogisticRegression(penalty = 'none') # solver default is lbfgs
log_fit_equal = regressor.fit(X_1_2_equal, y_1_2_equal)

cvs(log_fit_equal, X_1_2_equal, y_1_2_equal,cv=5)

# accuracy score for each split
```

```
## array([0.65      , 0.5       , 0.45      , 0.64102564, 0.43589744])
```

###2.2.iv. Do L2-regularisation with the following `Cs=  [1e5, 1e1, 1e-5]`. Use the same kind of cross-validation as in Exercise 2.2.iii. In the best-scoring of these models, how many more/fewer predictions are correct (on average)?

```
# Same as above penalty = L2, logistic regression as a C- argument (opposite of lambda)
# do a for loop, for C in....

Cs=  [1e5, 1e1, 1e-5]

for c in Cs:
  log = LogisticRegression(penalty = 'l2', C=c) # solver default is lbfgs
  log_fit_equal = log.fit(X_1_2_equal, y_1_2_equal)
  scores = cvs(log_fit_equal, X_1_2_equal, y_1_2_equal,cv=5)
  print(scores.mean())


# Amount of predictions correct
```

```
## 0.5353846153846155
## 0.5252564102564102
## 0.5956410256410256
```

```
from sklearn.model_selection import cross_val_predict as cvp
log_c_neg5 = LogisticRegression(penalty='l2', C=1e-5)
predict_c_neg5 = cvp(log_c_neg5, X_1_2_equal, y_1_2_equal, cv=5)
accuracy_neg5 = predict_c_neg5 == y_1_2_equal
## this is from Mina and I don't quite understand it, *** Write to Mina and ask :)

## Accuracy Log Model 2.2iii
predict_log = cvp(regressor, X_1_2_equal, y_1_2_equal, cv=5)
accuracy_log = predict_log == y_1_2_equal
print("Correct Predictions Log 2.2iii:", len(np.where(accuracy_log == True)[0]))
```

```
## Correct Predictions Log 2.2iii: 106
```

```
print("Correct Predictions Log 1e-5:", len(np.where(accuracy_neg5 == True)[0]))
```

```
## Correct Predictions Log 1e-5: 118
```

Based on the scores, Cs of 1e-5 is the most accurate, 60%, where the other two were 53% and 54%. We also have more correct predictions with the penalized model.

###2.2.v. Instead of fitting a model on all `n_sensors * n_samples` features, fit a logistic regression (same kind as in Exercise 2.2.iv (use the `C` that resulted in the best prediction)) for **each** time sample and use the same cross-validation as in Exercise 2.2.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?)

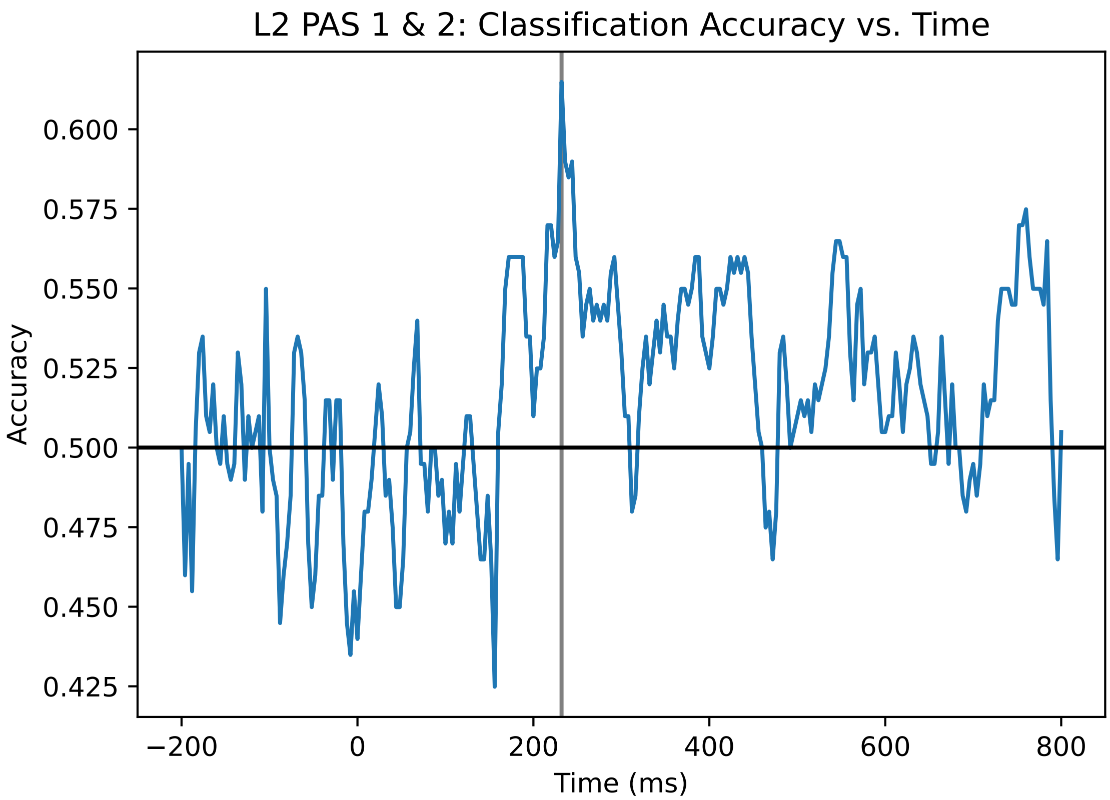
```
## empty list for the cross scores
cross_scores = []

for i in range(251):
  #Creating data and scaling
  scaler = StandardScaler()
  X_time = data_1_2_equal[:,:,i]
  X_time_scaled = scaler.fit_transform(X_time)

#Creating a logistic regression object
```

```
  lr = LogisticRegression(penalty='l2', C=1e-5)

#Cross-validating
  score = cvs(lr, X_time_scaled, y_1_2_equal, cv = 5)

#taking the mean
  mean = np.mean(score)

#appending the mean
  cross_scores.append(mean)

  #print(cross_scores)
# on knit, don't include printed output
```

```
## FINDING the time point where classification is best ##
indexmax = cross_scores.index(max(cross_scores))
times[indexmax]
```

```
## 232
```

```
plt.figure()
plt.axvline(x = times[indexmax], color = "black", alpha = 0.5)
plt.plot(times, cross_scores)
plt.axhline(y = 0.50, color = "black")
plt.title("L2 PAS 1 & 2: Classification Accuracy vs. Time")
plt.xlabel("Time (ms)")
plt.ylabel("Accuracy")
plt.show()
```



The chance level is .5 or 50% because it is a binary classification, either it's pas 1/pas2 or not.

###2.2.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)?

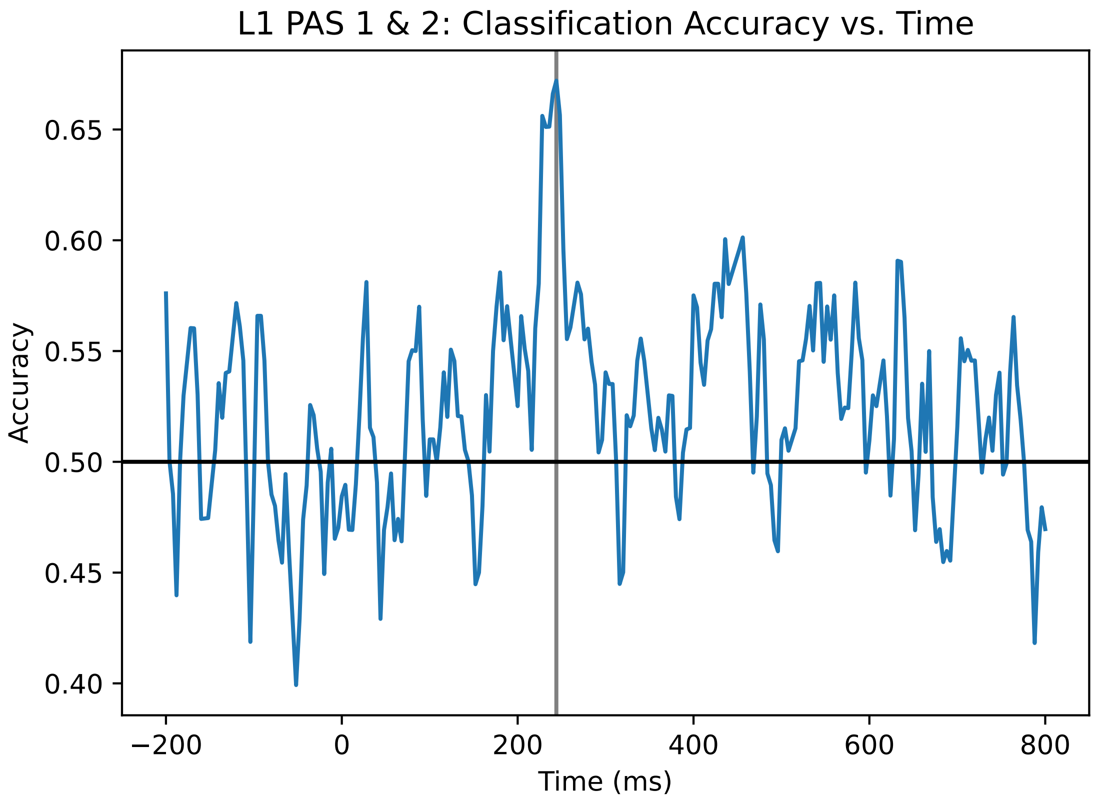
```
cross_scores_l1 = []

for i in range(251):
  #Creating data and scaling
  scaler = StandardScaler()
  X_time = data_1_2_equal[:,:,i]
  X_time_scaled = scaler.fit_transform(X_time)
  logr = LogisticRegression(penalty='l1', solver = "liblinear", C=1e-1)
  score = cvs(logr, X_time_scaled, y_1_2_equal, cv = 5)
  mean = np.mean(score)
  cross_scores_l1.append(mean)
```

```
indexmax_l1 = cross_scores_l1.index(max(cross_scores_l1))
times[indexmax_l1]
```

```
## 244
```

```
plt.figure()
plt.axvline(x = times[indexmax_l1], color = "black", alpha = 0.5)
plt.plot(times, cross_scores_l1)
plt.axhline(y = 0.50, color = "black")
plt.title("L1 PAS 1 & 2: Classification Accuracy vs. Time")
```

```
plt.xlabel("Time (ms)")
plt.ylabel("Accuracy")
plt.show()
```



###2.2.vii. Finally, fit the same models as in Exercise 2.2.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?)

```
pas14 = np.where((y == 1) | (y == 4))

data_1_4 = data[pas14] # np.squeeze gets rid of the point that is only 1

data_1_4.shape
```

```
## (359, 102, 251)
```

```
data_1_4.ndim # how many dimensions
```

```
## 3
```

```
y_1_4 = np.squeeze(y[pas14])

len(y_1_4)

# equalize the data
```

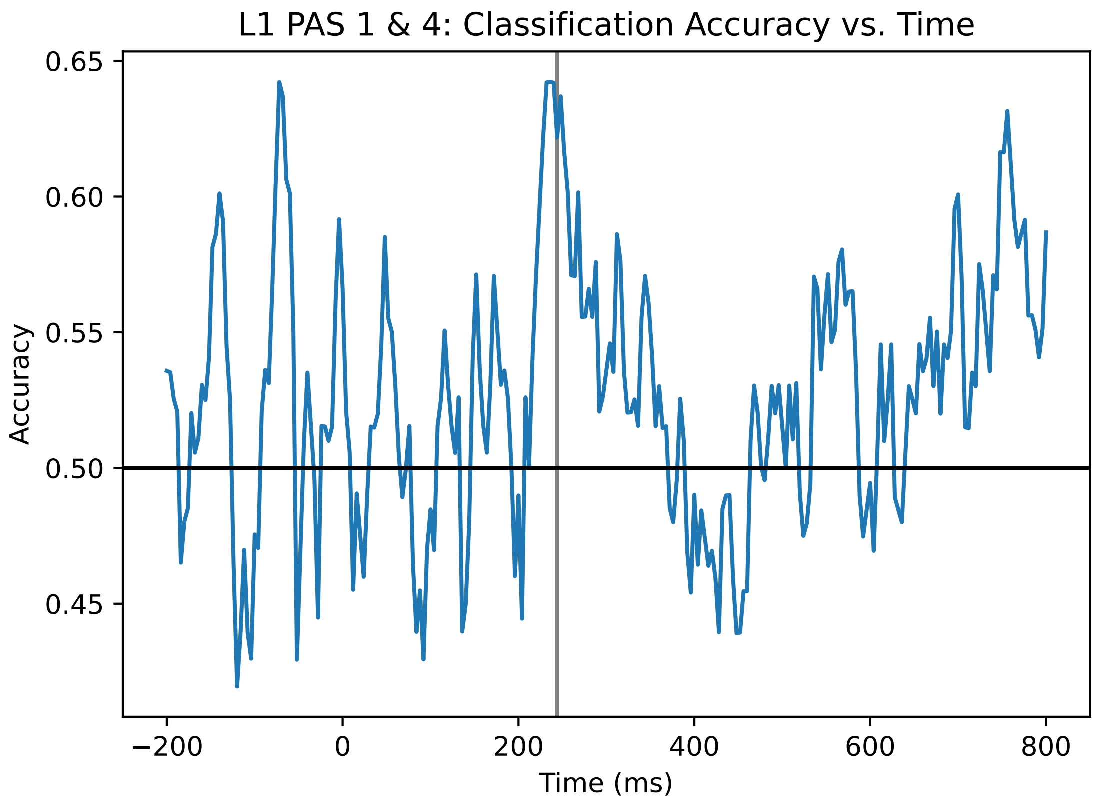
```
## 359
```

```
data_1_4_equal, y_1_4_equal = equalize_targets_binary(data_1_4, y_1_4)


cross_scores_pas14 = []

for i in range(251):
  #Creating data and scaling
  scaler = StandardScaler()
  X_time = data_1_4_equal[:,:,i]
  X_time_scaled = scaler.fit_transform(X_time)
  logr = LogisticRegression(penalty='l1', solver = "liblinear", C=1e-1)
  score = cvs(logr, X_time_scaled, y_1_4_equal, cv = 5)
  mean = np.mean(score)
  cross_scores_pas14.append(mean)
```

```
indexmax_pas14 = cross_scores_pas14.index(max(cross_scores_pas14))
times[indexmax_pas14]
```

```
## 236
```

```
plt.figure()
plt.axvline(x = times[indexmax_l1], color = "black", alpha = 0.5)
plt.plot(times, cross_scores_pas14)
plt.axhline(y = 0.50, color = "black")
plt.title("L1 PAS 1 & 4: Classification Accuracy vs. Time")
plt.xlabel("Time (ms)")
plt.ylabel("Accuracy")
plt.show()
```



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

- Expect to see more of a difference between pas 1 and 4 than pas 1 and 2 but we don't.
- That the accuracy is only a little higher than chance.

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

```
# Standard scale before making model, especially with linear model
# see emil's slides
```

##3.1) Do a Support Vector Machine Classification
###3.1.i. First equalize the number of targets using the function associated with each PAS-rating using the function associated with Exercise 3.1.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

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

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

```
y_equal.shape

#transform data to 2d
#data_equal = data_equal.reshape(396, -1)
```

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

```
X_equal = data_equal.reshape(data_equal.shape[0],-1)
X_equal.shape


# scale data
```

```
## (396, 25602)
```

```
scaler = StandardScaler()
X_equal_scale = scaler.fit_transform(X_equal)
X_equal_scale.shape
```

```
## (396, 25602)
```

###3.1.ii. Run two classifiers, one with a linear kernel and one with a radial basis (other options should be left at their defaults) - the number of features is the number of sensors multiplied the number of samples. Which one is better predicting the category?

```
from sklearn.svm import SVC
svm_linear = SVC(kernel ='linear')
svm_radial = SVC(kernel ='rbf')

# cross validating the linear support vector #
svm_linear_scores = cvs(svm_linear, X_equal_scale, y_equal, cv=5)
# cross validating the radial support vector #
svm_radial_scores = cvs(svm_radial, X_equal_scale, y_equal, cv=5)
## printing the mean of the cross-validated performances ##
print("SVM Linear Mean Cross Validated:", round(np.mean(svm_linear_scores), 3))
```

```
## SVM Linear Mean Cross Validated: 0.293
```

```
print("SVM Radial Mean Cross Validated:", round(np.mean(svm_radial_scores), 3))
```

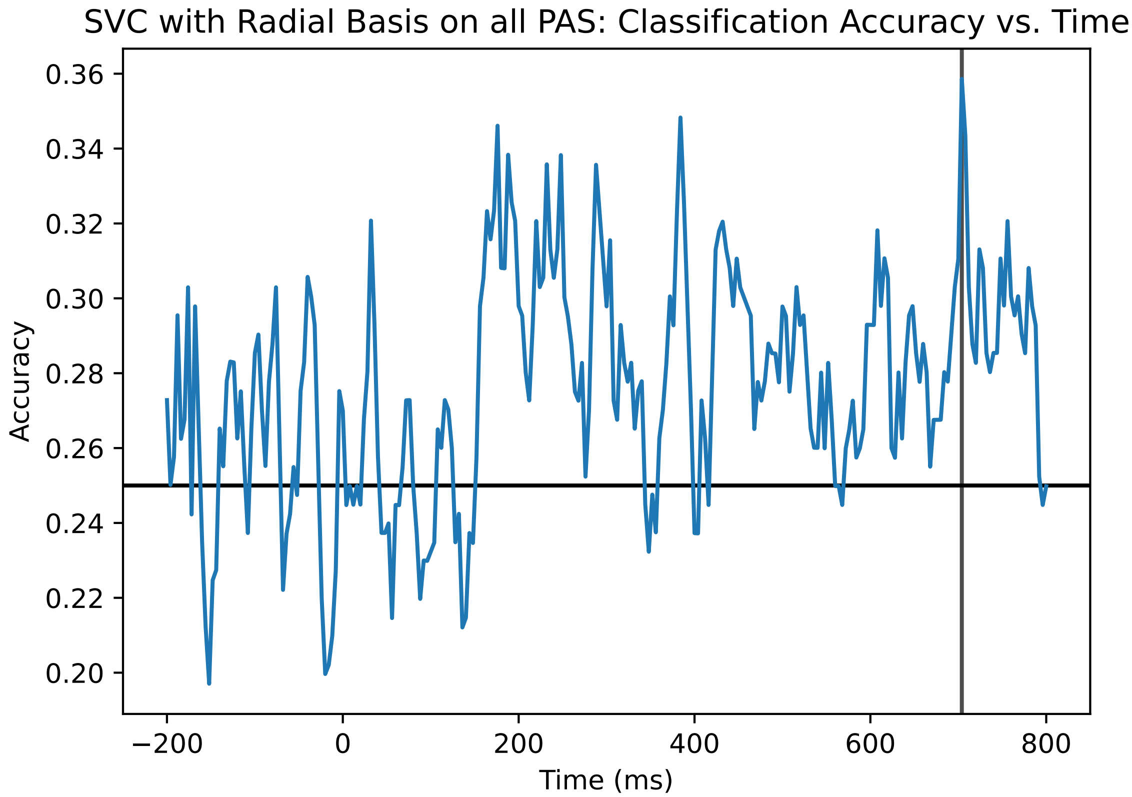
```
## SVM Radial Mean Cross Validated: 0.333
```

The radial support machine vector is more accurate but both have a fairly low accuracy. They are both above chance level though, which is 25% in this case with four pas options. ###3.1.iii. Run the sample-by-sample analysis (similar to Exercise 2.2.v) with the best kernel (from Exercise 3.1.ii). 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?)

```
# EMPTY list for the cross scores #
cross_scores_svm_radial = []

for i in range(251):
  #Creating data and scaling
  scaler = StandardScaler()
  X_time = data_equal[:, :, i]
  X_time_scale = scaler.fit_transform(X_time)

  #Instantiating a support vector machine with a radial basis
  svm_radial = SVC(kernel ='rbf')

  #Cross-validating
  score = cvs(svm_radial, X_time_scale, y_equal, cv = 5)

  #taking the mean
  mean = np.mean(score)

  #appending the mean
  cross_scores_svm_radial.append(mean)

# Plotting it
indexRsvc = cross_scores_svm_radial.index(max(cross_scores_svm_radial))
times[indexRsvc]
```

```
## 704
```

```
plt.figure()
plt.axvline(x = times[indexRsvc], color = "black", alpha = 0.7)
plt.axhline(y = 0.25, color = "black") #horizontal line at chance level
plt.plot(times, cross_scores_svm_radial)
plt.title("SVC with Radial Basis on all PAS: Classification Accuracy vs. Time")
plt.xlabel("Time (ms)")
plt.ylabel("Accuracy")
plt.show()
```



###3.1.iv. Is classification of subjective experience possible at around 200-250 ms? - yes, because it's above the chance level but still not very accurate.

##3.2) Finally, split the equalized data set (with all four ratings) into a training part and test part, where the test part if 30 % of the trials. Use `train_test_split` from `sklearn.model_selection`

```
# x_train, x_test, y_train, y_test = train_test_split(data, y, test_size, random_state = 12)
# random_state so numbers don't change every time

#Importing package
import sklearn.model_selection as sklearn

# Splitting into training and test parts
x_train, x_test, y_train, y_test = sklearn.train_test_split(X_equal, y_equal, test_size=0.3, random_stat
print(x_train)
```

```
## [[ 1.45955188e-15  4.20492290e-14  2.57370252e-14 ... -1.17676237e-12
##   -1.20117968e-12 -1.20603709e-12]
##  [ 1.63957549e-13  1.31537428e-13  9.65870848e-14 ...  4.47480062e-14
##    4.52375032e-14  1.10408835e-14]
##  [ 8.54634018e-14  3.04650996e-14 -3.21188347e-14 ... -7.77991915e-14
##   -1.42165865e-13 -2.00435884e-13]
##  ...
##  [-1.21069954e-13 -1.21170797e-13 -1.54069842e-13 ... -2.11593989e-13
##   -2.33431798e-13 -2.28392931e-13]
##  [ 4.65651023e-14  4.52869764e-14  5.77643727e-14 ... -5.08015067e-13
##   -5.17357994e-13 -5.38200566e-13]
```

```
## [ 5.70982874e-14  5.27696383e-14  2.35376281e-14 ... -5.29012874e-13
##   -5.36936517e-13 -5.46280337e-13]]
```

```
print(x_test)
```

```
## [[ 1.06186343e-13  1.38548039e-13  1.36509949e-13 ... -2.71679513e-13
##   -2.85633316e-13 -3.06449382e-13]
##  [ 1.81570845e-13  5.61135729e-14 -4.21174108e-14 ... -2.70918602e-13
##   -2.91448819e-13 -2.30672016e-13]
##  [-8.51434090e-14 -7.11056555e-14 -3.71459167e-14 ...  1.55338487e-14
##    2.05561868e-15  2.47112044e-14]
##  ...
##  [-9.22745614e-14 -9.33893468e-14 -7.17963163e-14 ... -1.72574404e-13
##   -2.93171422e-13 -3.59791245e-13]
##  [ 1.30355516e-13  1.34005597e-13  1.31952273e-13 ... -3.59074304e-13
##   -3.43736210e-13 -2.90459564e-13]
##  [ 1.63454033e-13  2.05205249e-13  2.03952393e-13 ...  8.42849807e-13
##    8.22806755e-13  7.73230914e-13]]
```

```
print(y_train)
```

```
## [4 3 4 1 2 1 3 2 1 3 1 4 3 4 3 4 3 2 2 3 1 1 2 1 4 4 2 4 1 1 1 1 1 4 1 1 2
##  4 3 4 4 1 3 3 1 1 1 1 3 1 4 4 3 2 4 4 3 2 2 2 3 2 3 4 4 1 4 2 3 1 3 1 3 2
##  4 3 1 4 3 3 3 1 2 4 4 2 1 2 1 1 2 2 4 4 2 2 4 3 1 1 1 1 3 3 2 3 4 4 1 2 4
##  1 3 4 2 3 2 3 2 4 1 3 4 1 3 1 1 1 4 2 1 2 1 2 1 2 3 4 4 3 1 4 2 2 2 1 1 3
##  2 3 4 3 3 4 2 1 2 3 3 4 3 4 4 2 1 4 2 3 1 2 1 1 2 2 2 3 1 1 4 4 1 4 2 1 3
##  2 2 1 3 2 2 2 3 4 4 4 1 2 1 2 4 2 3 4 4 4 2 2 2 1 3 3 2 2 1 1 1 2 4 4 4 3
##  2 1 2 2 4 4 2 4 3 3 4 1 3 4 3 3 1 4 4 2 1 2 2 3 2 1 2 2 2 3 2 4 2 2 2 3 1
##  1 3 3 2 4 1 2 2 1 3 4 3 2 3 3 4 2 4]
```

```
print(y_test)
```

```
## [3 4 1 1 4 3 3 3 3 2 1 2 1 4 3 4 3 1 1 3 3 1 4 4 4 2 1 4 2 1 3 4 4 3 1 3 2
##  4 1 3 4 1 2 4 3 4 3 2 3 2 4 1 3 2 4 3 2 1 3 4 2 2 3 1 2 4 3 4 2 4 3 1 4 4
##  1 4 1 3 3 1 2 2 2 3 1 3 4 3 4 3 3 2 1 2 1 1 3 1 3 4 4 4 2 3 4 2 1 4 4 1 3
##  3 3 3 1 1 2 4 1]
```

###3.2.i. Use the kernel that resulted in the best classification in Exercise 3.1.ii and `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.

```
svm_radial.fit(x_train, y_train)
```

```
## SVC()
```

```
predicted_y = svm_radial.predict(x_test)
print(predicted_y)
```

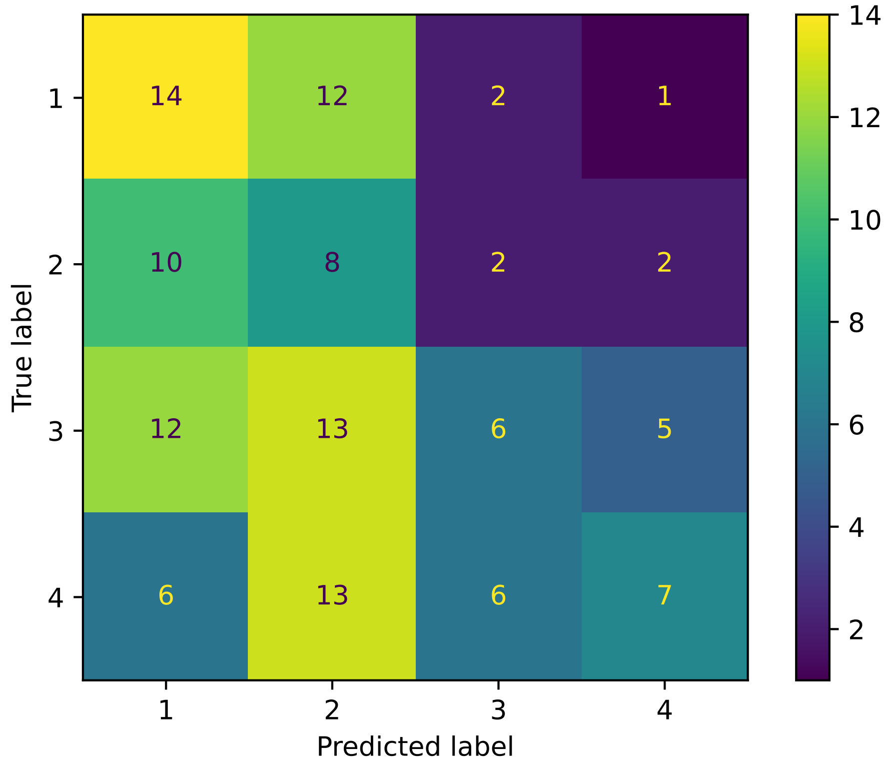
```
## [1 4 3 1 4 2 3 2 2 1 1 1 2 2 3 2 1 1 1 4 1 2 2 2 2 1 2 4 1 1 3 1 1 4 1 1 1
##  2 4 2 1 2 2 3 1 4 1 2 1 2 3 2 4 4 2 2 1 1 2 3 4 2 2 2 3 4 1 2 1 1 4 2 3 2
##  1 1 2 2 2 1 3 2 1 1 2 2 1 2 2 2 2 1 1 2 1 1 4 1 3 3 4 2 2 1 3 2 2 2 4 3 1
##  1 3 3 2 1 1 2 2]
```

###3.2.ii. 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 th e 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

```
from sklearn.metrics import ConfusionMatrixDisplay
ConfusionMatrixDisplay.from_predictions(y_test, predicted_y)
```

```
## <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay object at 0x16dd53e50>
```

```
plt.show()
```



###3.2.iii. 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?
- For PAS1-ratings half of the predtictions were correctly classified. It makes good sense that most of the predictions for PAS1 lies within PAS 1 and 2, as PAS 3 and 4 means a very clear experience for the participants. - The classifier is defnitely biased towards PAS 1 and 2. - Overall, the classifier does not perform very well. This, though, makes good sense as the radial bases kernel had an accuracy score of only 33%. Both the radial bases and the linear kernel classifiers had poor accuracy rates — but the radial bases kernel performed best and thus, we chose to use this one.