Portfolio 3 , Methods 3, 2021, autumn semester

Author: Sigurd Fyhn Sørensen

Date: 17-11-21

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)

In [ ]:
```
import numpy as np
import pandas as pd
import scipy as sp
import matplotlib.pyplot as plt
```

In [ ]:
```
#LOAD DATA ONLINE
""" import requests
import io

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


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

Out[ ]:
```
" import requests\nimport io\n\nresponse = requests.get('http://laumollerander
sen.org/data_methods_3/megmag_data.npy')\nresponse.raise_for_status()\ndata =
np.load(io.BytesIO(response.content))\n\n\nresponse = requests.get('http://lau
```

molleranderesen.org/data_methods_3/pas_vector.npy')\nresponse.raise_for_status()\ny = np.load(io.BytesIO(response.content)) "

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
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?

In [ ]:

```
#load data local
data = np.load("/Users/sigurd/Downloads/megmag_data.npy")
y = np.load("/Users/sigurd/Downloads/pas_vector.npy")
```

```
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.
```

In [ ]:

```
time = np.arange(-200,804, 4)
```

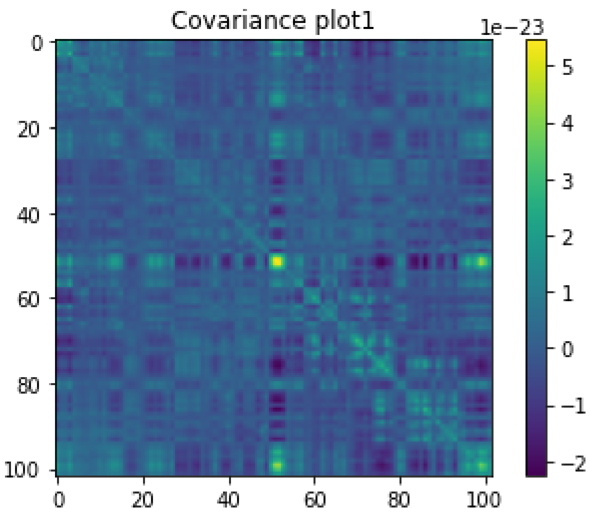
```
iii. Create the sensor covariance matrix $\Sigma_{XX}$:
$$\Sigma_{XX} = \frac 1 N \sum_{i=1}^N XX^T$$ $N$ is the number
of repetitions and $X$ has $s$ rows and $t$ columns (sensors and
time), thus the shape is $X_{s\times t}$. Do the sensors pick up
independent signals? (Use `plt.imshow` to plot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

In [ ]:

```
#1/682 * np.sum(data[0,:,:]@data[0,:,:].T)
cov_matrix = 1/682 * sum([data[x,:,:]@data[x,:,:].T for x in range(682)])
plt.imshow(cov_matrix)
plt.title("Covariance plot1")
plt.colorbar()
```

Out[ ]:

```
<matplotlib.colorbar.Colorbar at 0x7fe1c63aa130>
```



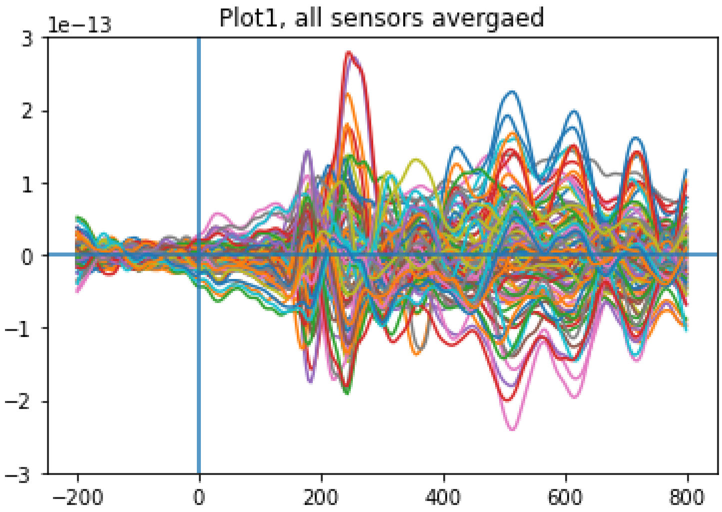
Sensors seem to pick up independent signals, but we cannot be sure there is some covariance between a few of the sensors.

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)

In [ ]:
```
rep_mean = np.mean(data, axis = 0) #axis specifies which axis the mean should
```

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`

In [ ]:
```
for i in range(102):
    plt.plot(time , rep_mean[i,:])
plt.axvline(0)
plt.axhline(0)
plt.title("Plot1, all sensors avergaed")
plt.ylim(-3e-13, 3e-13)
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.

In [ ]:
```
indx_max = np.unravel_index(np.argmax(rep_mean), rep_mean.shape)

print("mean:", rep_mean[indx_max], "\nindx:", indx_max)
```

```
mean: 2.7886216843591933e-13
indx: (73, 112)
```

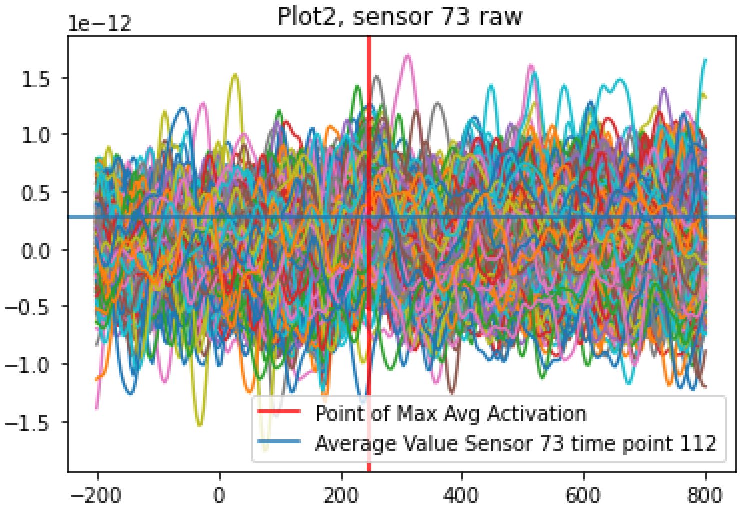
Sensor = 73 at time point = 112 has the highest value. This would be the red peak around 220ms on the above plot.

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`
```

In [ ]:

```python
data[:,73,:].shape

for i in range(682):
    plt.plot(time, data[i,73,:])
plt.axvline(112*4-200, color = "red", label = "Point of Max Avg Activation")
plt.axhline(rep_mean[73,112], label = "Average Value Sensor 73 time point 112
plt.title("Plot2, sensor 73 raw")
plt.legend()
plt.show()
```



Notice the decimal change on the y-axis. Plot1's Y scale is 1e-13 and plot2 1e-12. On plot2 0.2788 which look to be the mean around the hline would be equal to 2.788 in plot1 on the scale of 1e-e13.

```
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
```

Mean value at the time stamp 220ms across all repetitions = 2.7886216843591933e-13. This is roughly the same value we see in the plot illustrating the averages across repetitions. However, there is a lot of noise surrounding the mean signal. The Std for timestamp = 220ms is even higher than the mean. We must therefore conclude that there is a lot of noise.

In [ ]:

```python
print('std:', np.std(data[:,73,112]))
print('mean:', np.mean(data[:,73,112]))
```

```
std: 3.189439776492671e-13
mean: 2.7886216843591933e-13
```

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

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

It has the same length as the first dimension of our ndarray with MEG data. This must be because that there is an individual score for each repetition.

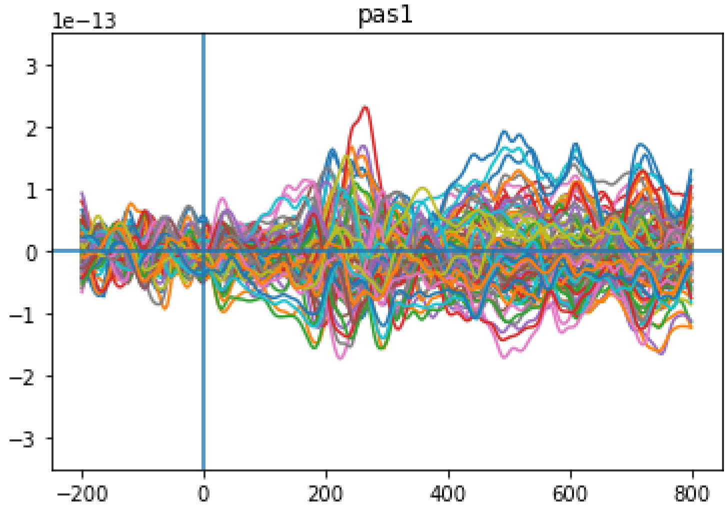
ii. Now make four averages (As in Exercise 1.1.iii), one for each PAS rating, and plot the four time courses (one for each PAS rating) for the sensor found in Exercise 1.1.v

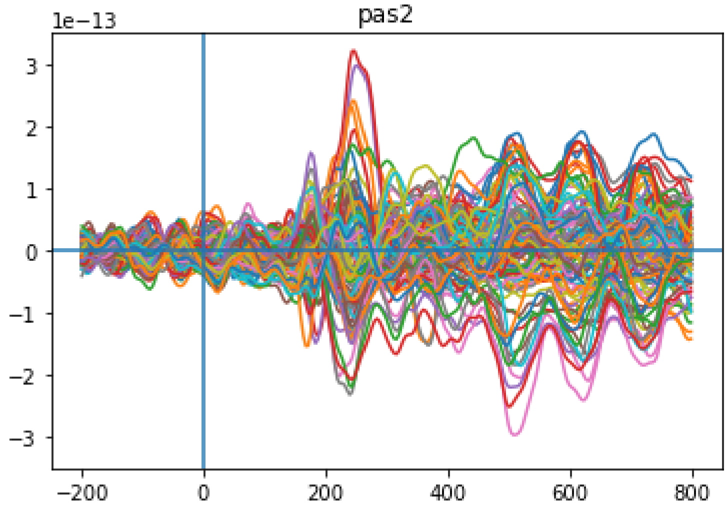
In [ ]:

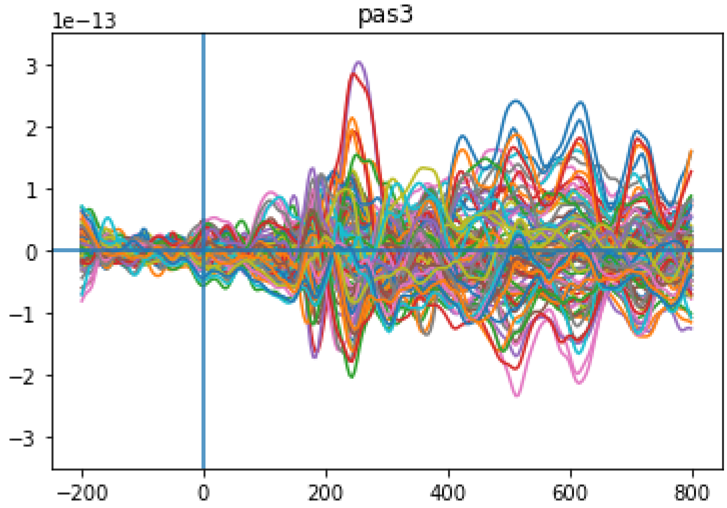
```
d = {}
#Create a dictonary with 4 levels on for each pas rating containing the data
for i in range(1,5):
    idx_ = np.argwhere(y == i)
    d["pas" + str(i)] = np.squeeze(data[idx_, : , :])
```

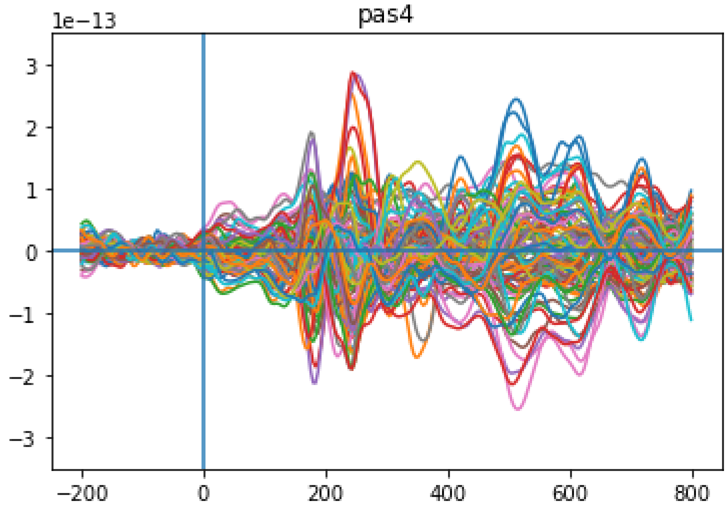
In [ ]:

```
rep_mean_dic = {}
#Compute the mean across itterations for each PAS rating.  Save in a dict wit
for key in d:
    rep_mean_dic[str(key)] = np.mean(d[key], axis = 0)

#Plot the avg for each PAS.
def plot_pas_rat():
    for key in rep_mean_dic:
        temp = rep_mean_dic[key]
        for i in range(102):
            plt.plot(time , temp[i,:])
        plt.axvline(0)
        plt.ylim(-3.5e-13, 3.5e-13)
        plt.axhline(0)
        plt.title(str(key))
        plt.show()
plot_pas_rat()
```



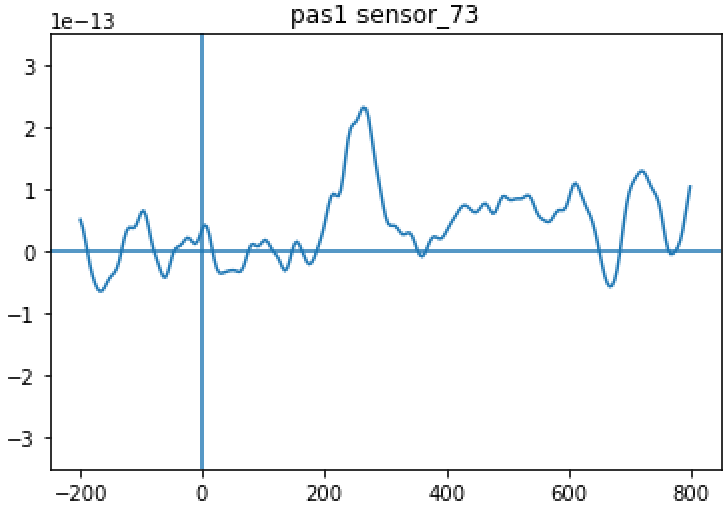


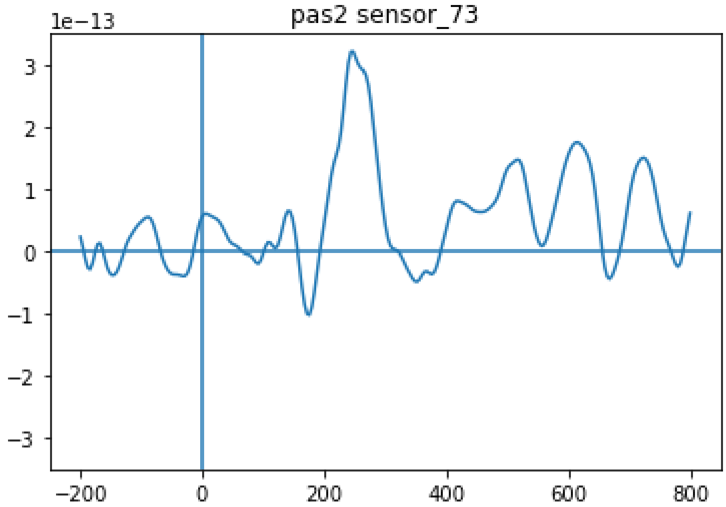


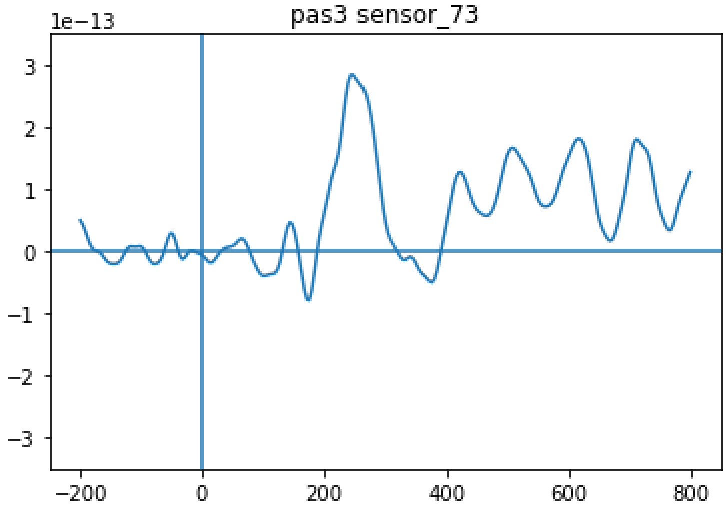


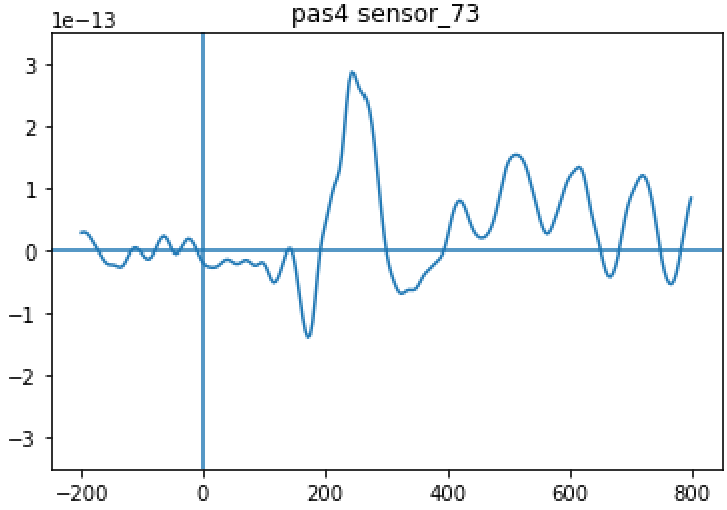
In [ ]:

```
for key in rep_mean_dic:
    temp = rep_mean_dic[key]

    plt.plot(time , temp[73,:])
    plt.axvline(0)
    plt.ylim(-3.5e-13, 3.5e-13)
    plt.axhline(0)
    plt.title(str(key) + " sensor_73")
    plt.show()
```


pas1 sensor_73
1e−13
3
2
1
0
−1
−2
−3
−200
0
200
400
600
800

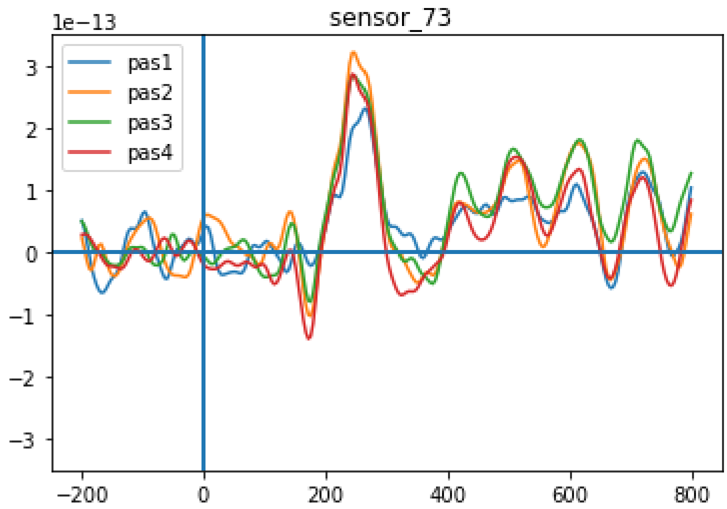

pas2 sensor_73
1e−13
3
2
1
0
−1
−2
−3
−200
0
200
400
600
800


pas3 sensor_73
1e−13
3
2
1
0
−1
−2
−3
−200
0
200
400
600
800



In [ ]:

```
for key in rep_mean_dic:
    temp = rep_mean_dic[key]

    plt.plot(time , temp[73,:], label = str(key))
    plt.axvline(0)
    plt.ylim(-3.5e-13, 3.5e-13)
    plt.axhline(0)
    plt.title(" sensor_73")
    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?
```

Higher amplitudes indicate a shift in magnetic fields, which is correlated with brain activity/ hemoglobin increase at target areas. I don't specifically where sensor 73 were placed. But it seems a higher PAS-score results in a higher brain response in the given area.

PAS2 The amplitude before 200ms seem to become more negative the higher the PAS score which is followed by the maximum around 250ms that increases with PAS-rating. However, comparing PAS-rating = 2 at sensor 73 to PAS-rating = 2|3, PAS2 seem to have a higher amplitude at 250ms and lower around 180ms.

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

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

```
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.
```

In [ ]:
```
idx = np.argwhere((y ==1) | (y ==2))
data_1_2 = np.squeeze(data[idx,:,:])
y_1_2 = y[idx]
y_1_2 = np.reshape(y_1_2, (len(y_1_2)))

X_1_2 = np.reshape(data_1_2, newshape = (data_1_2.shape[0], data_1_2.shape[1]
```

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

In [ ]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_1_2_std = sc.fit_transform(X_1_2)
```

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

In [ ]:
```
from sklearn.linear_model import LogisticRegression
lgR_1 = LogisticRegression(max_iter= 1000, penalty= 'none')
#fit
lgR_1.fit(X_1_2_std, y_1_2)
```

Out[ ]:
```
LogisticRegression(max_iter=1000, penalty='none')
```

```
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?
```

In [ ]:
```
lgR_1.score(X_1_2_std, y_1_2)
```

Out[ ]:
```
1.0
```

We get an accuracy score of 1 which indicates overfitting (or a perfect model if it could generalize to out-of-sample prediction). Another indication that we could be overfitting would be our lack of penalty in our model.

```
vi. Now apply the _L1_ penalty instead - how many of the
coefficients (`.coef_`) are non-zero after this?
```

In [ ]:

```
lgR_2 = LogisticRegression(penalty = 'l1', solver = "liblinear", C = 1 , max_
lgR_2.fit(X_1_2_std, y_1_2)
```

Out[ ]:

```
LogisticRegression(C=1, max_iter=400, penalty='l1', solver='liblinear')
```

In [ ]:

```
print("accuracy score %f" % (lgR_2.score(X_1_2_std, y_1_2)*100), "%")
print("non-zero coef = %f" % np.sum(lgR_2.coef_ != 0))
print("number of  zero coef = %f"  % np.sum(lgR_2.coef_ == 0))
```

```
accuracy score 100.000000 %
non-zero coef = 194.000000
number of  zero coef = 25408.000000
```

10997 coefficients are non-zero when using l1 regularization/penalizing.

```
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
$X_{reduced}^TX_{reduced}$ (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?
```

In [ ]:

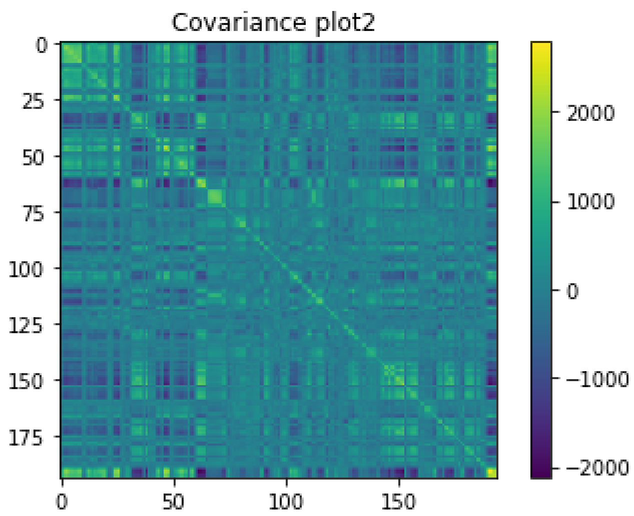
```
idx_non0 = np.where(lgR_2.coef_ != 0)[1]
data_reduced = X_1_2_std[:,idx_non0]
```

In [ ]:

```
cov = data_reduced.T @ data_reduced
plt.imshow(np.cov(cov))
plt.title("Covariance plot2 ")
plt.colorbar()
```

Out[ ]:

```
<matplotlib.colorbar.Colorbar at 0x7fe1a2f5f5e0>
```



There is less covariance in plot 2 compared to covariance plot 1. It is difficult to quantify these differences just by visual inspection, but it gives a good intuition.

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`

In [ ]:
```
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
```

```
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`!
```

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

X_1_2_equa , y_1_2_equa  = equalize_targets_binary(data_1_2, y_1_2)
X_1_2_equa_2d = X_1_2_equa.reshape(198,-1)


sc = StandardScaler()
X_1_2_equa_std = sc.fit_transform(X_1_2_equa_2d)

#X_1_2_equa_std = sc.fit_transform(X_1_2_equa)
```

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

In [ ]:
```
kfold = StratifiedKFold(n_splits= 5, random_state = 2, shuffle= True).split(X_

scores = []
lr = LogisticRegression(penalty= "l1", solver = "liblinear")

for k, (train, test) in enumerate(kfold):
    lr.fit(X_1_2_equa_std[train], y_1_2[train])
    score = lr.score(X_1_2_equa_2d[test], y_1_2_equa[test])
```

```
    scores.append(score)
    print('Fold: %2d, Class dist.: %s, Acc: %.3f' % (k+1,
        np.bincount(y_1_2[train]), score))

cv_score = cross_val_score(lr, X_1_2_equa_std, y_1_2_equa, cv= 5)
print("accuracy scores for k-folds:" , cv_score)
```

```
Fold:  1, Class dist.: [ 0 69 89], Acc: 0.500
Fold:  2, Class dist.: [ 0 69 89], Acc: 0.500
Fold:  3, Class dist.: [ 0 71 87], Acc: 0.500
Fold:  4, Class dist.: [ 0 74 85], Acc: 0.487
Fold:  5, Class dist.: [ 0 73 86], Acc: 0.513
accuracy scores for k-folds: [0.625      0.575      0.425      0.64102564 0.46
153846]
```

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)?

In [ ]:

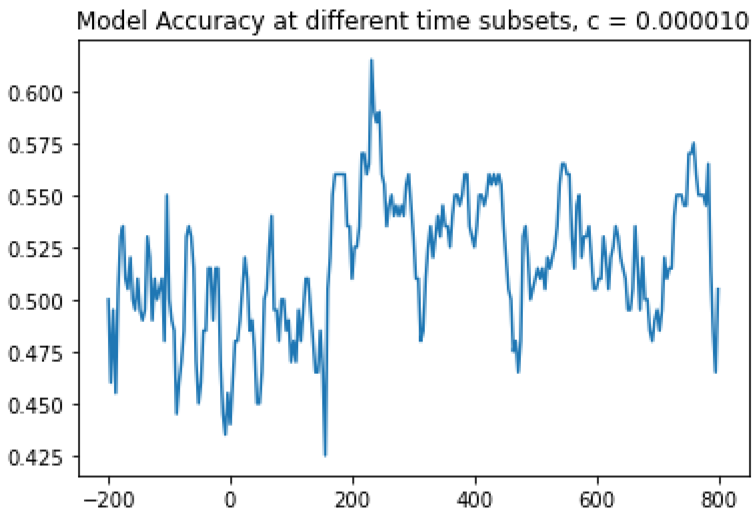
```
def cv_different_C(a):
    for i in a:
        cv_score = []
        lr2 = LogisticRegression(penalty= "l2", C = i)
        cv_score = cross_val_score(lr2, X_1_2_equa_std, y_1_2_equa, cv = 5)
        print("mean accuracy for c =",str(i)+ ":", np.mean(cv_score))

cv_different_C([1e5, 1e1, 1e-5])
```

```
mean accuracy for c = 100000.0: 0.5353846153846155
mean accuracy for c = 10.0: 0.5252564102564102
mean accuracy for c = 1e-05: 0.5956410256410256
```

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?)

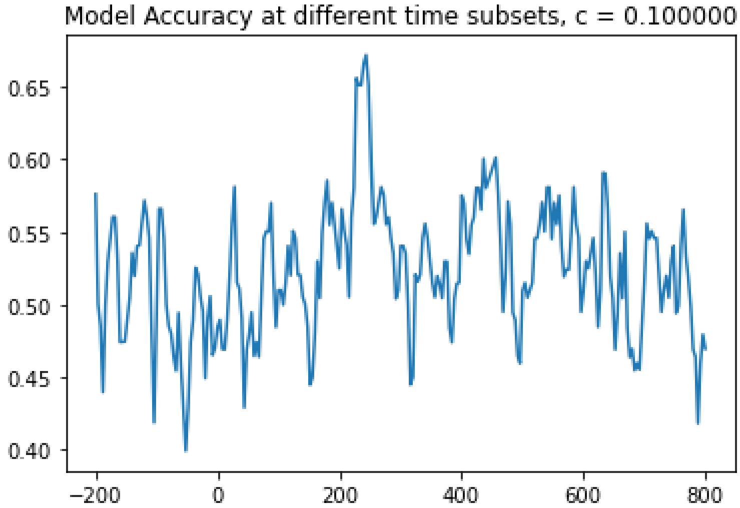
In [ ]:

```
def cv_different_C_i(a, penal, solver_lul, X, y):
    accuracy = []
    lr3 = LogisticRegression(penalty = penal, C = a, solver = solver_lul)
    for i in range(251):
        cv_score = []
        sc = StandardScaler()
        X_std = sc.fit_transform(X[:,:,i])
        cv_score = cross_val_score(lr3, X_std, y)
        accuracy.append(np.mean(cv_score))
    return(accuracy)

def plot_cv_time(a, penal, solver_lul, X , y ):
    cv_acc_means = cv_different_C_i(a, penal, solver_lul, X , y)
    plt.plot(time, cv_acc_means)
    plt.title("Model Accuracy at different time subsets, c = %f" % a )
    plt.show()

plot_cv_time(a = 1e-5, penal = "l2", solver_lul= "lbfgs", X = X_1_2_equa, y =
```



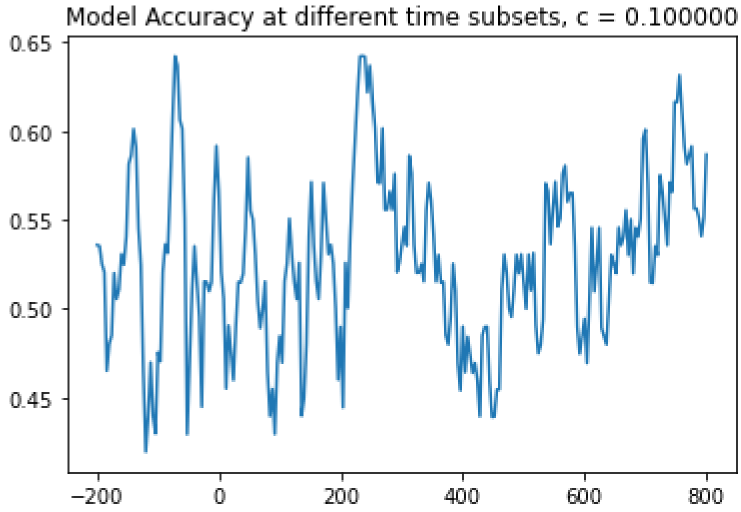
vi. Now do the same, but with L1 regression - set `C=1e-1` - what are the time points when classification is best? (make a plot)?

In [ ]:
```
plot_cv_time(a = 1e-1, penal = "l1", solver_lul= "liblinear", X = X_1_2_equa,
```



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?)

In [ ]:
```
idx = np.argwhere((y ==1) | (y ==4)) # Get index for pas 1 & 4
data_1_4 = np.squeeze(data[idx,:,:]) #Squeeze the data and filter repitition b
y_1_4 = y[idx] #filter y by idx
y_1_4 = np.reshape(y_1_4, (len(y_1_4))) #get the right y_shape
X_1_4 = np.reshape(data_1_4, newshape = (data_1_4.shape[0], data_1_4.shape[1]
data_1_4_equa , y_1_4_equa = equalize_targets_binary(data_1_4, y_1_4) #equali



plot_cv_time(a = 1e-1, penal = "l1", solver_lul= "liblinear", X = data_1_4_eq
```



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?

Pairwise classification of PAS scores does not seem to be possible. Surprisingly the accuracy for PAS1-2 classification is higher compared to PAS1-2. This indicates that the difference between PAS1-2 is larger or more linearly separable than PAS1-4. But in general the accuracy is not much better than chance.

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

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

In [ ]:

```
def equalize_targets(data, y):
    np.random.seed(7)
    targets = np.unique(y) ## find the number of targets
    if len(targets) > 4:
        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
    third_choice = np.random.choice(indices[2], size = min_count, replace = F
    fourth_choice = np.random.choice(indices[3], size = min_count, replace = 
    # create the new data sets
    new_indices = np.concatenate((first_choice, second_choice, third_choice, 
    new_y = y[new_indices]
    new_data = data[new_indices, :, :]

    return new_data, new_y
```

In [ ]:

```
data_equa, y_equa = equalize_targets(data, y)
```

In [ ]:
```
sum(y_equa == 1), sum(y_equa == 2), sum(y_equa == 3), sum(y_equa == 4)
```

Out[ ]:
```
(99, 99, 99, 99)
```

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?

In [ ]:
```
data_equa_2d = np.reshape(data_equa, newshape= (data_equa.shape[0], -1))
data_equa_2d_std = sc.fit_transform(data_equa_2d)
```

In [ ]:
```
from sklearn.svm import SVC
from sklearn import svm
#Linear function
sv = SVC(kernel='linear')
sv.fit(data_equa_2d_std, y_equa)
```

Out[ ]:
```
SVC(kernel='linear')
```

In [ ]:
```
#radial basis function
sv2 = SVC(kernel = "rbf")
sv2.fit(data_equa_2d_std, y_equa)
```

Out[ ]:
```
SVC()
```

In [ ]:
```
print("score for linear function:",sv.score(data_equa_2d_std, y_equa), "\n sco
```

```
score for linear function: 1.0
 score for rbf: 0.9873737373737373
```

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?)

In [ ]:
```
def cv_different_svm(X, y):
    accuracy = []
    lr3 = SVC(kernel = "rbf", random_state= 10)
    for i in range(251):
        cv_score = []
        sc = StandardScaler()
        X_std = sc.fit_transform(X[:,:,i])
        cv_score = cross_val_score(lr3, X_std, y)
        accuracy.append(np.mean(cv_score))
        print("append:", i)
    return(accuracy)


svm_time_score = cv_different_svm(data_equa, y_equa)
```

```
append: 0
append: 1
append: 2
append: 3
append: 4
append: 5
append: 6
append: 7
append: 8
append: 9
append: 10
append: 11
append: 12
append: 13
append: 14
append: 15
append: 16
append: 17
append: 18
append: 19
append: 20
append: 21
append: 22
append: 23
append: 24
append: 25
append: 26
append: 27
append: 28
append: 29
append: 30
append: 31
append: 32
append: 33
append: 34
append: 35
append: 36
append: 37
append: 38
append: 39
append: 40
append: 41
append: 42
append: 43
append: 44
append: 45
append: 46
append: 47
append: 48
append: 49
append: 50
append: 51
append: 52
append: 53
append: 54
append: 55
append: 56
append: 57
append: 58
append: 59
append: 60
append: 61
append: 62
append: 63
```

```
append: 64
append: 65
append: 66
append: 67
append: 68
append: 69
append: 70
append: 71
append: 72
append: 73
append: 74
append: 75
append: 76
append: 77
append: 78
append: 79
append: 80
append: 81
append: 82
append: 83
append: 84
append: 85
append: 86
append: 87
append: 88
append: 89
append: 90
append: 91
append: 92
append: 93
append: 94
append: 95
append: 96
append: 97
append: 98
append: 99
append: 100
append: 101
append: 102
append: 103
append: 104
append: 105
append: 106
append: 107
append: 108
append: 109
append: 110
append: 111
append: 112
append: 113
append: 114
append: 115
append: 116
append: 117
append: 118
append: 119
append: 120
append: 121
append: 122
append: 123
append: 124
append: 125
append: 126
append: 127
```

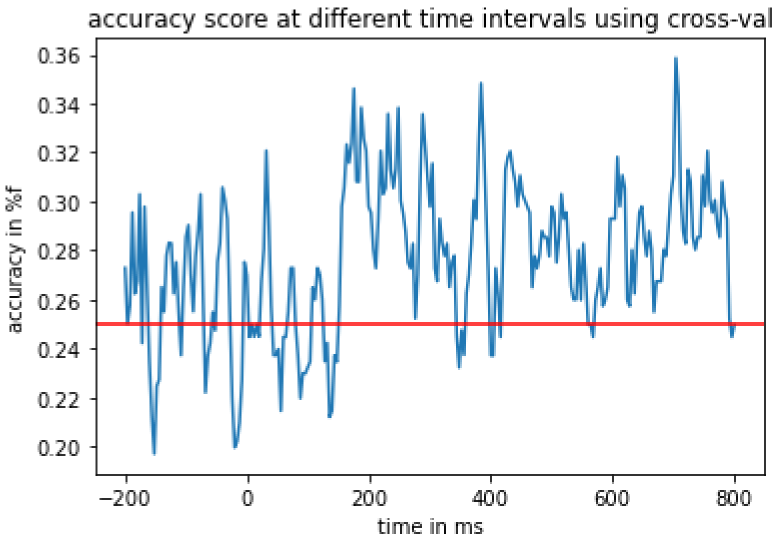
```
append: 128
append: 129
append: 130
append: 131
append: 132
append: 133
append: 134
append: 135
append: 136
append: 137
append: 138
append: 139
append: 140
append: 141
append: 142
append: 143
append: 144
append: 145
append: 146
append: 147
append: 148
append: 149
append: 150
append: 151
append: 152
append: 153
append: 154
append: 155
append: 156
append: 157
append: 158
append: 159
append: 160
append: 161
append: 162
append: 163
append: 164
append: 165
append: 166
append: 167
append: 168
append: 169
append: 170
append: 171
append: 172
append: 173
append: 174
append: 175
append: 176
append: 177
append: 178
append: 179
append: 180
append: 181
append: 182
append: 183
append: 184
append: 185
append: 186
append: 187
append: 188
append: 189
append: 190
append: 191
```

```
append: 192
append: 193
append: 194
append: 195
append: 196
append: 197
append: 198
append: 199
append: 200
append: 201
append: 202
append: 203
append: 204
append: 205
append: 206
append: 207
append: 208
append: 209
append: 210
append: 211
append: 212
append: 213
append: 214
append: 215
append: 216
append: 217
append: 218
append: 219
append: 220
append: 221
append: 222
append: 223
append: 224
append: 225
append: 226
append: 227
append: 228
append: 229
append: 230
append: 231
append: 232
append: 233
append: 234
append: 235
append: 236
append: 237
append: 238
append: 239
append: 240
append: 241
append: 242
append: 243
append: 244
append: 245
append: 246
append: 247
append: 248
append: 249
append: 250
```

In [ ]:

```
plt.plot(time, svm_time_score)
plt.axhline(0.25, color = "red")
plt.xlabel("time in ms")
plt.ylabel("accuracy in %f")
```

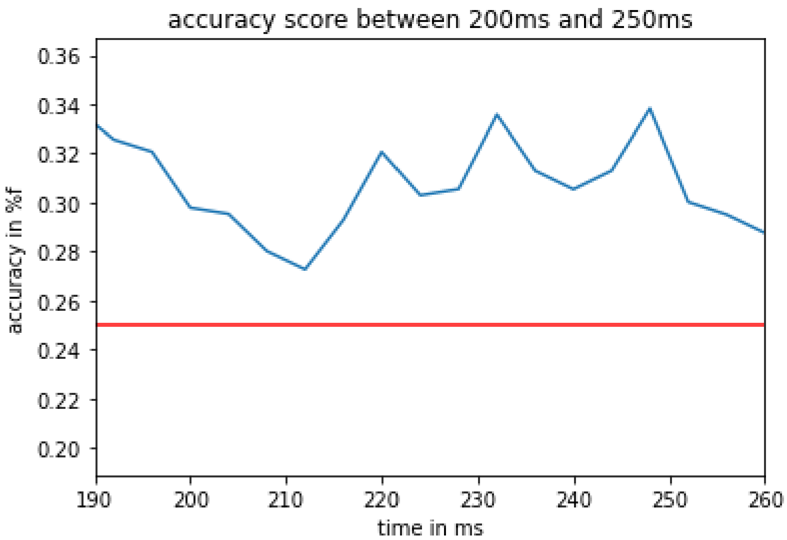
```
plt.title("accuracy score at different time intervals using cross-val")
plt.show()
```



iv. Is classification of subjective experience possible at around 200-250 ms?

In [ ]:

```
plt.plot(time, svm_time_score)
plt.axhline(0.25, color = "red")
plt.xlim(190,260)
plt.xlabel("time in ms")
plt.ylabel("accuracy in %f")
plt.title("accuracy score between 200ms and 250ms")
plt.show()
```



Our support vector machine classification performs better than chance (25%) when applied sequentially to the frames between 200 ms and 250ms. The max accuracy of our sequential modelling of PAS rating does not excede 33% at any time point. I would therefore argue that with our current model a classification of subjective experience in the time span between 200-250 ms is not possible.

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`

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.

In [ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data_equa, y_equa, test_s
#Reshape
X_train_2d = np.reshape(X_train, newshape= (X_train.shape[0], -1))
X_test_2d = np.reshape(X_test, newshape= (X_test.shape[0], -1))

X_train_2d = sc.fit_transform(X_train_2d)
X_test_2d = sc.transform(X_test_2d)


sv3 = SVC(kernel='linear', random_state= 10)
sv4 = SVC(kernel = "rbf", random_state= 10)
sv3.fit(X_train_2d, y_train)
sv4.fit(X_train_2d, y_train)

y_pred_svm_split = sv3.predict(X_test_2d)
y_pred_svm2_split = sv4.predict(X_test_2d)
```

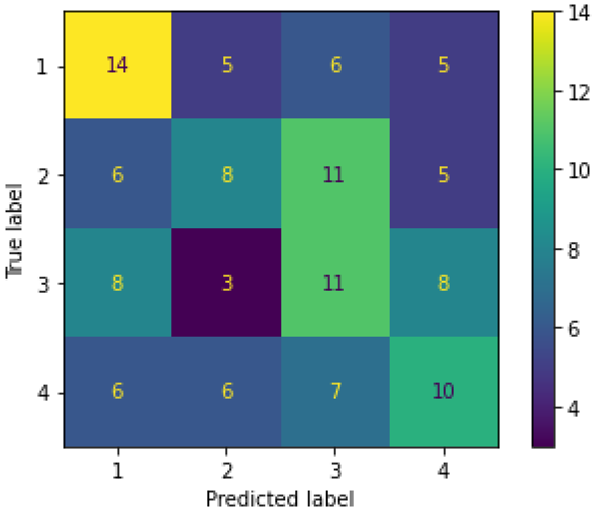
In [ ]:
```
print("linear score: " , sv3.score(X_test_2d, y_test), "\n rbf score:" ,sv4.s
```

```
linear score:  0.36134453781512604
 rbf score: 0.35294117647058826
```

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 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

In [ ]:
```
from sklearn.metrics import confusion_matrix , ConfusionMatrixDisplay
#Confusion matrix for linaer kernel
confmat = confusion_matrix(y_test, y_pred_svm_split, labels= sv3.classes_)
confmat_disp = ConfusionMatrixDisplay(confmat, display_labels= sv3.classes_)
confmat_disp.plot()
```

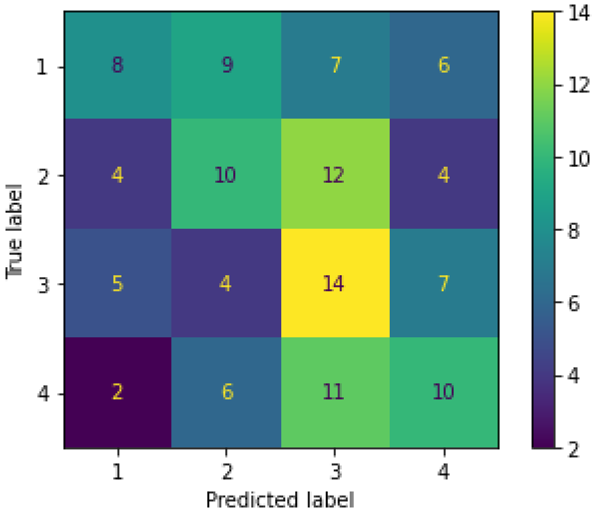
Out[ ]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7fe19ccab7
90>
```



In [ ]:

```
confmat = confusion_matrix(y_test, y_pred_svm2_split, labels= sv4.classes_)
confmat_disp = ConfusionMatrixDisplay(confmat, display_labels= sv4.classes_)
confmat_disp.plot()
```

Out[ ]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7fe1c78ae520>




```
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?
```

Linear kernel

36.13% accuracy. It chooses a PAS rating as being the dominant. With the current random_state it is PAS2. But with other states it chooses randomly. We can therefore conclude that the guessing is more up to chance than to observed regularities. Since the features of PAS 2,3,4 is so similar our linear SVM really struggles with distinguishing between those.

RBF kernel

35.29% accuracy. You would hope that RBF SVM were able to distinguish between the non-linear separable data. Changing the random_state again results in a change of PAS bias. So it seems that even introducing non-linear dimensionality expansion through the RBF kernel does not supply anything to the model that would be much better than just chance. This is also supported by our relatively low accuracy scores.

Getting a better accuracy with deep learning NN

While SVM and LogisticRegression has its usefulness it is also restricted to only having one layer of tensor operations. Deep learning has proven useful when dealing with large amount of structured data when classifying perceptual tasks. Our data is stored in a 3D tensor (sample, sensor/feature, time) for dealing with such data the default architecture is a convolutional network or the long-short-term-memory (LSTM) network. But for now let us just work with stackable Dense layers and the same tensor shape as fed to the SVM and Logistic classifiers.

In [ ]:

```
import keras
from keras.models import Sequential
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import np_utils
from sklearn.model_selection import cross_val_score , KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from keras.regularizers import l2
import tensorflow as tf
from keras.layers import LSTM, Dropout , Dense
from sklearn.decomposition import PCA
```

In [ ]:

```
#load data local
data = np.load("/Users/sigurd/Downloads/megmag_data.npy")
y = np.load("/Users/sigurd/Downloads/pas_vector.npy")
```

Dense NN

In [ ]:

```
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(y_equa)
encoded_Y = encoder.transform(y_equa)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
```

In [ ]:

```
pc = PCA(n_components= 30)
data_equa_pca_2d_std = pc.fit_transform(data_equa_2d_std)
data_equa_pca_2d_std.shape
```

Out[ ]:

```
(396, 30)
```

In [ ]:

```
#Model fit without function
model = Sequential()
model.add(Dense(16, input_dim = 30, activation='relu'))
#model.add(layer= layers.Conv2D( 16 , 8 , padding='same', activation='relu'))
model.add(Dense(16, activation = 'relu', kernel_regularizer= l2(0.01), bias_r
model.add(Dense(4, activation='softmax'))
```

```
# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['ac
```

Train, Train_val and Test split

Note to self: The test split should actually only be standardized and transformed based on the parameters of the train data split. Right now we've some data leakage between the test and train splits, but we will ignore it for now.

In [ ]:

```
from sklearn.model_selection import train_test_split
train_data, test_data, train_y, test_y = train_test_split(data_equa_pca_2d_st

data_val = train_data[:99]
data_train = train_data[99:]

y_val = train_y[:99]
y_train = train_y[99:]

"""from sklearn.preprocessing import Normalizer
scaler = Normalizer()

data_equa_2d_norm = scaler.fit_transform(data_equa_2d)
train_data2, test_data2, train_y2, test_y2 = train_test_split(data_equa_2d_no
```

Out[ ]:

```
'from sklearn.preprocessing import Normalizer\nscaler = Normalizer()\n\ndata_e
qua_2d_norm = scaler.fit_transform(data_equa_2d)\ntrain_data2, test_data2, tra
in_y2, test_y2 = train_test_split(data_equa_2d_norm, dummy_y, stratify= dummy_
y)'
```

In [ ]:

```
#train
model.fit(train_data, train_y, epochs = 10, batch_size= 8, verbose = 1, valid

#evalutate on test data
model.evaluate(test_data, test_y)
```

```
Epoch 1/10
38/38 [==============================] - 0s 4ms/step - loss: 0.6346 - accurac
y: 0.8249 - val_loss: 0.5574 - val_accuracy: 0.8384
Epoch 2/10
38/38 [==============================] - 0s 2ms/step - loss: 0.5722 - accurac
y: 0.8586 - val_loss: 0.5960 - val_accuracy: 0.8384
Epoch 3/10
38/38 [==============================] - 0s 2ms/step - loss: 0.5291 - accurac
y: 0.8923 - val_loss: 0.6141 - val_accuracy: 0.7980
Epoch 4/10
38/38 [==============================] - 0s 2ms/step - loss: 0.5015 - accurac
y: 0.8855 - val_loss: 0.6355 - val_accuracy: 0.7980
Epoch 5/10
38/38 [==============================] - 0s 2ms/step - loss: 0.4801 - accurac
y: 0.8990 - val_loss: 0.6648 - val_accuracy: 0.7677
Epoch 6/10
38/38 [==============================] - 0s 2ms/step - loss: 0.4652 - accurac
y: 0.9091 - val_loss: 0.6922 - val_accuracy: 0.7677
Epoch 7/10
38/38 [==============================] - 0s 2ms/step - loss: 0.4553 - accurac
y: 0.9091 - val_loss: 0.7270 - val_accuracy: 0.7475
Epoch 8/10
38/38 [==============================] - 0s 2ms/step - loss: 0.4402 - accurac
y: 0.9091 - val_loss: 0.7502 - val_accuracy: 0.7374
Epoch 9/10
```

```
38/38 [==============================] - 0s 2ms/step - loss: 0.4307 - accurac
y: 0.9226 - val_loss: 0.7961 - val_accuracy: 0.6869
Epoch 10/10
38/38 [==============================] - 0s 2ms/step - loss: 0.4215 - accurac
y: 0.9091 - val_loss: 0.7932 - val_accuracy: 0.6970
4/4 [==============================] - 0s 1ms/step - loss: 0.7932 - accuracy:
0.6970
```

Out[ ]:

```
[0.7932385802268982, 0.6969696879386902]
```

report first model

Our validation score: accuracy = 69.7%, loss = 0.79.

Test score: accuracy = 69.6%, loss = 0.79.

Our test and validations scores are fairly similair this may be due to the data leakage between train and test.

Cross-validation of a new model

In [ ]:

```
# define baseline model
def baseline_model():
        # create model
        model = Sequential()
        model.add(Dense(32, input_dim= 30, activation='relu'))
        #model.add(layer= layers.Conv2D( 16 , 8 , padding='same', activation=
        model.add(Dense(64, activation = 'relu'))
        model.add(Dense(32, activation = 'relu'))
        model.add(Dense(4, activation='softmax'))
        # Compile model
        model.compile(loss='categorical_crossentropy', optimizer='adam', metr
        return model
```

In [ ]:

```
estimator = KerasClassifier(build_fn= baseline_model , epochs=20, batch_size=
#kFold
kfold = KFold(n_splits=5, shuffle=True)
```

In [ ]:

```
#Cross val based on NN
results = cross_val_score(estimator, data_equa_pca_2d_std , dummy_y, cv=kfold
```

```
Epoch 1/20
22/22 [==============================] - 0s 1ms/step - loss: 9.7627 - accurac
y: 0.2216
Epoch 2/20
22/22 [==============================] - 0s 1ms/step - loss: 3.1832 - accurac
y: 0.2909
Epoch 3/20
22/22 [==============================] - 0s 1ms/step - loss: 2.2075 - accurac
y: 0.3404
Epoch 4/20
22/22 [==============================] - 0s 2ms/step - loss: 1.7746 - accurac
y: 0.4084
Epoch 5/20
22/22 [==============================] - 0s 2ms/step - loss: 1.3294 - accurac
y: 0.4993
Epoch 6/20
22/22 [==============================] - 0s 1ms/step - loss: 1.1060 - accurac
y: 0.5104
```

```
Epoch 7/20
22/22 [==============================] - 0s 1ms/step - loss: 1.0671 - accurac
y: 0.6102
Epoch 8/20
22/22 [==============================] - 0s 1ms/step - loss: 0.9191 - accurac
y: 0.6074
Epoch 9/20
22/22 [==============================] - 0s 1ms/step - loss: 0.8164 - accurac
y: 0.6619
Epoch 10/20
22/22 [==============================] - 0s 1ms/step - loss: 0.7483 - accurac
y: 0.7305
Epoch 11/20
22/22 [==============================] - 0s 1ms/step - loss: 0.6619 - accurac
y: 0.7649
Epoch 12/20
22/22 [==============================] - 0s 1ms/step - loss: 0.5972 - accurac
y: 0.7858
Epoch 13/20
22/22 [==============================] - 0s 1ms/step - loss: 0.5930 - accurac
y: 0.8123
Epoch 14/20
22/22 [==============================] - 0s 1ms/step - loss: 0.5725 - accurac
y: 0.8115
Epoch 15/20
22/22 [==============================] - 0s 983us/step - loss: 0.4772 - accura
cy: 0.8559
Epoch 16/20
22/22 [==============================] - 0s 992us/step - loss: 0.4313 - accura
cy: 0.8792
Epoch 17/20
22/22 [==============================] - 0s 1ms/step - loss: 0.4235 - accurac
y: 0.8825
Epoch 18/20
22/22 [==============================] - 0s 994us/step - loss: 0.3864 - accura
cy: 0.8951
Epoch 19/20
22/22 [==============================] - 0s 1ms/step - loss: 0.3203 - accurac
y: 0.9407
Epoch 20/20
22/22 [==============================] - 0s 984us/step - loss: 0.2998 - accura
cy: 0.9505
6/6 [==============================] - 0s 985us/step - loss: 3.0446 - accurac
y: 0.3000
Epoch 1/20
22/22 [==============================] - 0s 956us/step - loss: 8.3519 - accura
cy: 0.2242
Epoch 2/20
22/22 [==============================] - 0s 1ms/step - loss: 3.4763 - accurac
y: 0.3389
Epoch 3/20
22/22 [==============================] - 0s 1ms/step - loss: 2.2756 - accurac
y: 0.3927
Epoch 4/20
22/22 [==============================] - 0s 1ms/step - loss: 1.5508 - accurac
y: 0.4805
Epoch 5/20
22/22 [==============================] - 0s 1ms/step - loss: 1.3454 - accurac
y: 0.5501
Epoch 6/20
22/22 [==============================] - 0s 1ms/step - loss: 1.2797 - accurac
y: 0.5651
Epoch 7/20
22/22 [==============================] - 0s 1ms/step - loss: 1.0112 - accurac
```

```
y: 0.6400
Epoch 8/20
22/22 [==============================] - 0s 1ms/step - loss: 0.8030 - accurac
y: 0.7189
Epoch 9/20
22/22 [==============================] - 0s 1ms/step - loss: 0.5746 - accurac
y: 0.7966
Epoch 10/20
22/22 [==============================] - 0s 883us/step - loss: 0.6111 - accura
cy: 0.7833
Epoch 11/20
22/22 [==============================] - 0s 907us/step - loss: 0.5339 - accura
cy: 0.8386
Epoch 12/20
22/22 [==============================] - 0s 886us/step - loss: 0.4401 - accura
cy: 0.8901
Epoch 13/20
22/22 [==============================] - 0s 894us/step - loss: 0.5636 - accura
cy: 0.8018
Epoch 14/20
22/22 [==============================] - 0s 857us/step - loss: 0.4469 - accura
cy: 0.8665
Epoch 15/20
22/22 [==============================] - 0s 991us/step - loss: 0.3599 - accura
cy: 0.9119
Epoch 16/20
22/22 [==============================] - 0s 983us/step - loss: 0.3339 - accura
cy: 0.9297
Epoch 17/20
22/22 [==============================] - 0s 996us/step - loss: 0.2711 - accura
cy: 0.9568
Epoch 18/20
22/22 [==============================] - 0s 1ms/step - loss: 0.2519 - accurac
y: 0.9562
Epoch 19/20
22/22 [==============================] - 0s 990us/step - loss: 0.2121 - accura
cy: 0.9855
Epoch 20/20
22/22 [==============================] - 0s 936us/step - loss: 0.1976 - accura
cy: 0.9791
6/6 [==============================] - 0s 994us/step - loss: 3.0082 - accurac
y: 0.3165
Epoch 1/20
22/22 [==============================] - 0s 973us/step - loss: 6.5288 - accura
cy: 0.2171
Epoch 2/20
22/22 [==============================] - 0s 1ms/step - loss: 2.7782 - accurac
y: 0.2577
Epoch 3/20
22/22 [==============================] - 0s 1ms/step - loss: 1.7732 - accurac
y: 0.3914
Epoch 4/20
22/22 [==============================] - 0s 1ms/step - loss: 1.4443 - accurac
y: 0.4608
Epoch 5/20
22/22 [==============================] - 0s 1ms/step - loss: 1.3159 - accurac
y: 0.5156
Epoch 6/20
22/22 [==============================] - 0s 984us/step - loss: 1.0569 - accura
cy: 0.5583
Epoch 7/20
22/22 [==============================] - 0s 1ms/step - loss: 1.0550 - accurac
y: 0.6079
Epoch 8/20
```

```
22/22 [==============================] - 0s 1ms/step - loss: 0.9140 - accurac
y: 0.6560
Epoch 9/20
22/22 [==============================] - 0s 1ms/step - loss: 0.7555 - accurac
y: 0.7307
Epoch 10/20
22/22 [==============================] - 0s 1ms/step - loss: 0.7269 - accurac
y: 0.7297
Epoch 11/20
22/22 [==============================] - 0s 1ms/step - loss: 0.6280 - accurac
y: 0.7840
Epoch 12/20
22/22 [==============================] - 0s 1ms/step - loss: 0.6283 - accurac
y: 0.8036
Epoch 13/20
22/22 [==============================] - 0s 1ms/step - loss: 0.5324 - accurac
y: 0.8265
Epoch 14/20
22/22 [==============================] - 0s 1ms/step - loss: 0.5077 - accurac
y: 0.8560
Epoch 15/20
22/22 [==============================] - 0s 985us/step - loss: 0.4486 - accura
cy: 0.8902
Epoch 16/20
22/22 [==============================] - 0s 1ms/step - loss: 0.3926 - accurac
y: 0.9358
Epoch 17/20
22/22 [==============================] - 0s 1ms/step - loss: 0.3816 - accurac
y: 0.9312
Epoch 18/20
22/22 [==============================] - 0s 954us/step - loss: 0.3242 - accura
cy: 0.9564
Epoch 19/20
22/22 [==============================] - 0s 1ms/step - loss: 0.3215 - accurac
y: 0.9362
Epoch 20/20
22/22 [==============================] - 0s 1ms/step - loss: 0.2933 - accurac
y: 0.9354
6/6 [==============================] - 0s 1ms/step - loss: 2.8747 - accuracy:
0.3544
Epoch 1/20
22/22 [==============================] - 0s 992us/step - loss: 8.7003 - accura
cy: 0.2243
Epoch 2/20
22/22 [==============================] - 0s 1ms/step - loss: 3.1328 - accurac
y: 0.2783
Epoch 3/20
22/22 [==============================] - 0s 1ms/step - loss: 2.1028 - accurac
y: 0.3245
Epoch 4/20
22/22 [==============================] - 0s 1ms/step - loss: 1.5756 - accurac
y: 0.4152
Epoch 5/20
22/22 [==============================] - 0s 1ms/step - loss: 1.3347 - accurac
y: 0.4942
Epoch 6/20
22/22 [==============================] - 0s 1ms/step - loss: 1.0977 - accurac
y: 0.5758
Epoch 7/20
22/22 [==============================] - 0s 1000us/step - loss: 1.0335 - accur
acy: 0.5943
Epoch 8/20
22/22 [==============================] - 0s 1ms/step - loss: 0.8614 - accurac
y: 0.6763
```

```
Epoch 9/20
22/22 [==============================] - 0s 976us/step - loss: 0.7685 - accura
cy: 0.7148
Epoch 10/20
22/22 [==============================] - 0s 1ms/step - loss: 0.6692 - accurac
y: 0.7864
Epoch 11/20
22/22 [==============================] - 0s 1ms/step - loss: 0.5924 - accurac
y: 0.8063
Epoch 12/20
22/22 [==============================] - 0s 966us/step - loss: 0.5502 - accura
cy: 0.8378
Epoch 13/20
22/22 [==============================] - 0s 1ms/step - loss: 0.5737 - accurac
y: 0.8201
Epoch 14/20
22/22 [==============================] - 0s 926us/step - loss: 0.4892 - accura
cy: 0.8710
Epoch 15/20
22/22 [==============================] - 0s 1ms/step - loss: 0.4142 - accurac
y: 0.9045
Epoch 16/20
22/22 [==============================] - 0s 922us/step - loss: 0.4273 - accura
cy: 0.8914
Epoch 17/20
22/22 [==============================] - 0s 944us/step - loss: 0.3666 - accura
cy: 0.9264
Epoch 18/20
22/22 [==============================] - 0s 898us/step - loss: 0.3616 - accura
cy: 0.9220
Epoch 19/20
22/22 [==============================] - 0s 920us/step - loss: 0.3205 - accura
cy: 0.9518
Epoch 20/20
22/22 [==============================] - 0s 926us/step - loss: 0.2651 - accura
cy: 0.9730
6/6 [==============================] - 0s 1ms/step - loss: 2.0605 - accuracy:
0.3797
Epoch 1/20
22/22 [==============================] - 0s 1ms/step - loss: 6.6385 - accurac
y: 0.2529
Epoch 2/20
22/22 [==============================] - 0s 1ms/step - loss: 3.3002 - accurac
y: 0.2869
Epoch 3/20
22/22 [==============================] - 0s 1ms/step - loss: 2.2598 - accurac
y: 0.3827
Epoch 4/20
22/22 [==============================] - 0s 1ms/step - loss: 1.5680 - accurac
y: 0.4756
Epoch 5/20
22/22 [==============================] - 0s 1ms/step - loss: 1.3596 - accurac
y: 0.5144
Epoch 6/20
22/22 [==============================] - 0s 1ms/step - loss: 1.0795 - accurac
y: 0.5901
Epoch 7/20
22/22 [==============================] - 0s 967us/step - loss: 0.8951 - accura
cy: 0.6590
Epoch 8/20
22/22 [==============================] - 0s 1ms/step - loss: 0.7758 - accurac
y: 0.6753
Epoch 9/20
22/22 [==============================] - 0s 968us/step - loss: 0.6115 - accura
```

```
cy: 0.7791
Epoch 10/20
22/22 [==============================] - 0s 944us/step - loss: 0.5326 - accura
cy: 0.8578
Epoch 11/20
22/22 [==============================] - 0s 982us/step - loss: 0.4658 - accura
cy: 0.8584
Epoch 12/20
22/22 [==============================] - 0s 899us/step - loss: 0.4256 - accura
cy: 0.8871
Epoch 13/20
22/22 [==============================] - 0s 922us/step - loss: 0.3642 - accura
cy: 0.9415
Epoch 14/20
22/22 [==============================] - 0s 1ms/step - loss: 0.3000 - accurac
y: 0.9698
Epoch 15/20
22/22 [==============================] - 0s 1ms/step - loss: 0.2693 - accurac
y: 0.9663
Epoch 16/20
22/22 [==============================] - 0s 1ms/step - loss: 0.3086 - accurac
y: 0.9497
Epoch 17/20
22/22 [==============================] - 0s 960us/step - loss: 0.2598 - accura
cy: 0.9646
Epoch 18/20
22/22 [==============================] - 0s 1ms/step - loss: 0.2018 - accurac
y: 0.9961
Epoch 19/20
22/22 [==============================] - 0s 1ms/step - loss: 0.2066 - accurac
y: 0.9794
Epoch 20/20
22/22 [==============================] - 0s 981us/step - loss: 0.1567 - accura
cy: 0.9931
6/6 [==============================] - 0s 993us/step - loss: 2.9845 - accurac
y: 0.3544
```

In [ ]:

```
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
Baseline: 34.10% (2.88%)
```

Report Cross-validated model

We created a new model with 1 additional hidden layer and larger tensor outputs which was test and validated through a 10-fold cross-validation. Features were standardized based on the Z distribution and extracted by PCA. For model hyperparameters a batch_size = 15 and epochs = 20 showed the best results. Our main metric of interest was accuracy with categorical cross entropy as our loss function.

The mean accuracy score across all 10-fold were 34.10% with a std = 2.88%. This is lower than our first model which was not cross-validated but only validated on a single split and tested on a final unseen split. This was expected due to CV being a stronger tool against overfitting. It is still better than chance level which is 25% 100*1/4 or see below chunk for exemplification of chance level. Right now we only have 396 samples which is quite small for a deep neural network. So by collecting additional data we could expect our classifier to improve significantly.

In [ ]:

```
import copy
```

```
chance_level = []
for i in range (100):
    test_labels_copy = copy.copy(y_equa)
    np.random.shuffle(test_labels_copy)
    hits_array = np.array(y_equa) == np.array(test_labels_copy)
    chance_level.append(float(np.sum(hits_array)) / len(y_equa))
np.mean(chance_level)
```

Out[ ]:
```
0.25
```