Practical 2

Task 1: The Data

We use the mouse protein expression dataset: https://archive.ics.uci.edu/ml/datasets/Mice+Protein+Expression Please use the code provided below for loading the dataset. Let's start with a bit of exploration.

Tasks 1.1: Dataset Exploration

- How many samples / features are provided?
- How many labels does the dataset have?
- What is the value range of the individuals predictors?
- Visualize the 10 first samples of the dataset in a form that highlights their differences.
- Visualize the variance of each predictor.

```
#from google.colab import drive
#drive.mount('/content/gdrive')
% cd gdrive/MyDrive/ml_2021
```

```
     /content/gdrive/MyDrive/ml_2021
```

```
import plotly as ply
import plotly.express as px
import plotly.graph_objects as go
from copy import deepcopy
```

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
```

```
import sklearn.linear_model as skl
```

```
file = 'data/Data_Cortex_Nuclear.csv'
df = pd.read_csv(file)

N = 10  # use only every tenth sample
X_all = df.iloc[::N,1:65].to_numpy()
t_all = (df['Behavior'] == 'S/C').to_numpy()[::N]

idx = ~np.any(np.isnan(X_all), axis=1)
X_all = X_all[idx]
t_all = t_all[idx]
```

```
#Include normalize (by range) function for plotting later
def normalize(data):
  cols = data.shape[1]

  #normdat = data
  normdat = deepcopy(data)

  for k in range(cols):
    mu = np.mean(normdat[:,k])
    max = np.max(normdat[:,k])
    min = np.min(normdat[:,k])

    normdat[:,k] = (normdat[:,k] - mu) / (max-min)

  return normdat
```

Find out answers to questions above:

```
#how many samples/features:
print("We've got", X_all.shape[0], "samples and", X_all.shape[1], "features.")
```

```
We've got 105 samples and 64 features.
```

```
#the classes/labels are in the 'class' variable
print("We've got", len(np.unique(t_all)), "different labels.")
```

```
We've got 2 different labels.
```

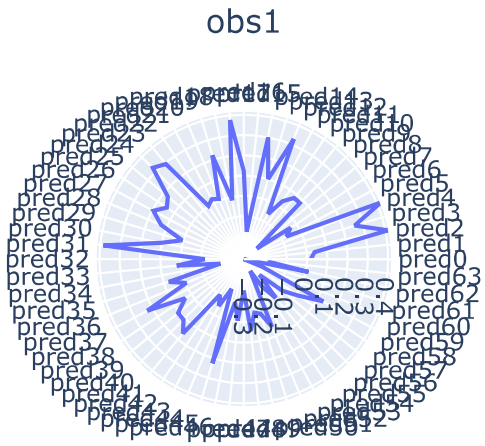
```
#print class labels
np.unique(df['Behavior'])
```

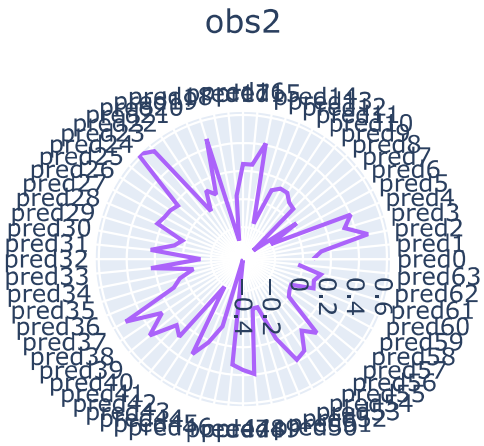
```
array(['C/S', 'S/C'], dtype=object)
```

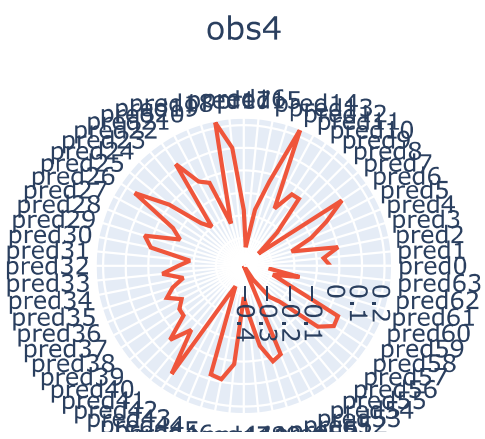
```
###ranges of numeric predictors

maxvals = np.max(X_all, axis = 0)
minvals = np.min(X_all, axis = 0)

const_dict = {'min': minvals, 'max': maxvals}
range_df = pd.DataFrame(const_dict)

#display range of first few predictors
range_df.head()
```

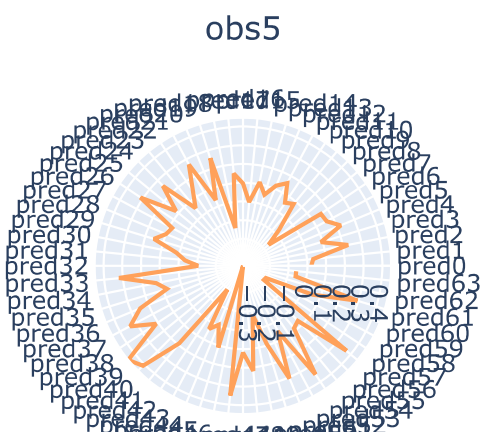
	min	max
0	0.168493	2.480316

```
##visualising first nine observations as this allows for a 3*3 grid
##and that just looks a lot nicer


normdat = normalize(X_all)

pred_names = ['pred'+str(i) for i in range(X_all.shape[1])]

#make plot
fig = ply.subplots.make_subplots(3, 3, specs=[[{'type': 'polar'}]*3]*3,
                                 horizontal_spacing = 0.2,
                                 vertical_spacing = 0.05,
                                 subplot_titles = ['obs'+str(i+1) for i in range(9)]
                                 )

for x in range(3):
  for y in range(3):
    k = x * 3 + y


    const_dict = {'theta': pred_names, 'r': normdat[k]}
    plot_df = pd.DataFrame(const_dict)

    fig.add_trace(go.Scatterpolar(
        r = plot_df['r'],
        theta = plot_df['theta'],
        mode = 'lines'
    ), row = y+1, col = x+1)

fig.update_layout(height=900, width=900, showlegend = False)
fig.show()
```

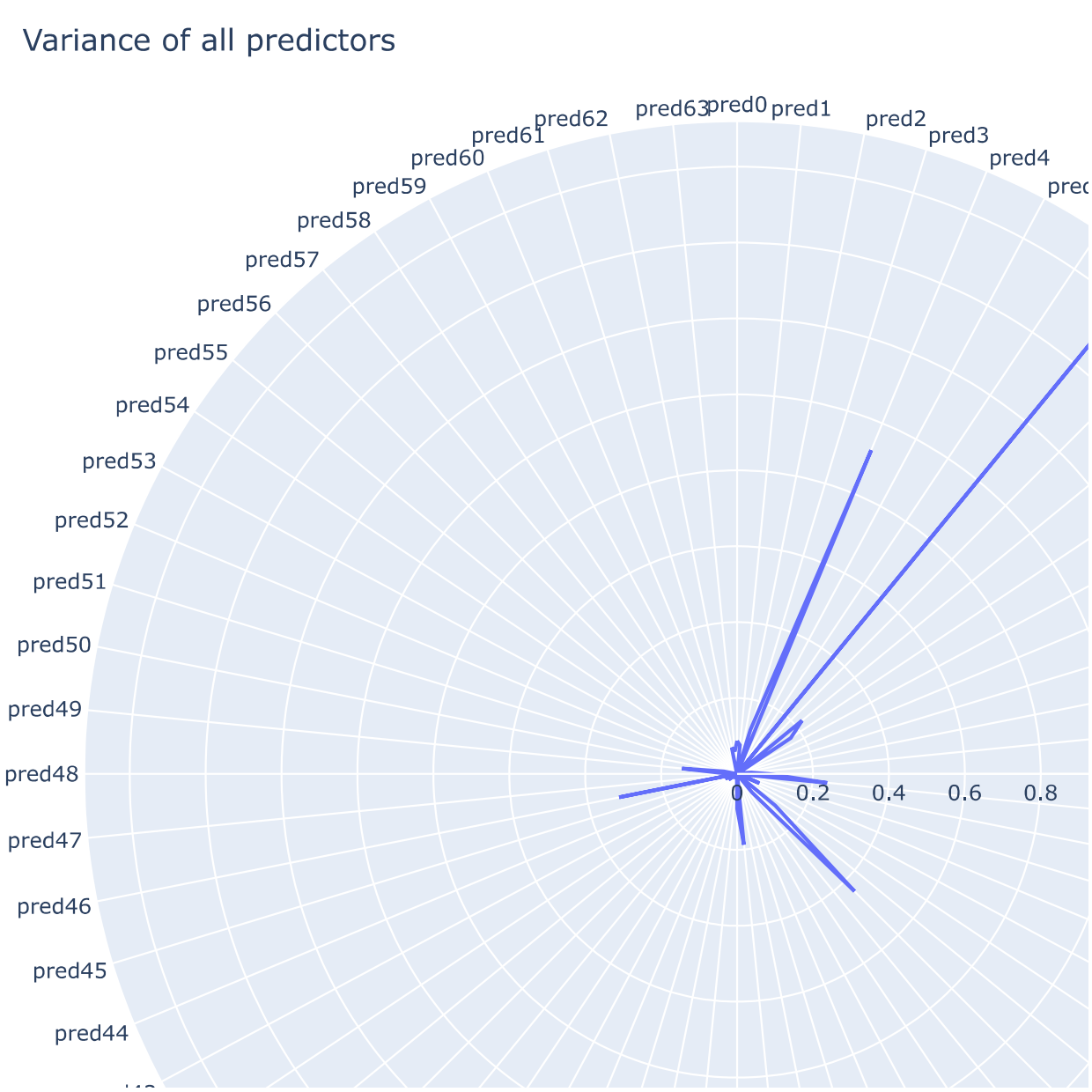









```
variances = np.var(X_all, axis = 0)

const_dict = {'var': variances, 'names': pred_names}
plot_df = pd.DataFrame(const_dict)


fig = px.line_polar(plot_df, r='var', theta='names', line_close=True)
fig.update_layout(height=900, width=900, showlegend = False, title = "Variance of all pred
fig.show()
```



Task 1.2: Data Preprocessing:

- Write a function `split_data(X, y, frac, seed)` that first shuffles your training data and then splits it into a training and a test set. `frac` determines the relative size of the test dataset, `seed` makes sure we get reproducible results.
- Write a function `preprocess(X)` which zero-centers your data and sets variance to one (per-feature).

```
def split_data(X, y, frac=0.3, seed=None):
    if seed is not None:
        np.random.seed(seed)

    ### implement the function here
    idx = np.arange(X.shape[0])
    idx_shuffled = np.random.permutation(idx)

    n_train = round(frac * X.shape[0])
    n_test = X.shape[0] - n_train

    # using the first n_train shuffled indexes for the training set
```

```
    # and the following observations for the training set
    idx_train = idx_shuffled[:n_train]
    idx_test = idx_shuffled[n_train:n_train + n_test]

    X_train = X[idx_train,:]
    X_test = X[idx_test,:]

    y_train = y[idx_train]
    y_test = y[idx_test]

    return X_train, y_train, X_test, y_test

def preprocess(X):
    X_norm = X.copy()
    for jj in range(0,X_norm.shape[1]):
        mean = np.mean(X_norm[:,jj])
        sd = np.std(X_norm[:,jj])

        X_norm[:,jj] = (X_norm[:,jj] - mean)/sd
    return X_norm
```

Task 2: LDA

First, use Linear Discriminant Analysis to separate the classes. As discussed in the Bishop in pg. 186-189, we can find a weight vector $\vec{w}$ that performs a projection of the i-th input data point $\vec{x}_i$

$$p = \vec{w}^T \vec{x}_i$$

that optimally separates the classes.

Use the analytic solution to compute the optimal weights $\vec{w}$ from the training set data.

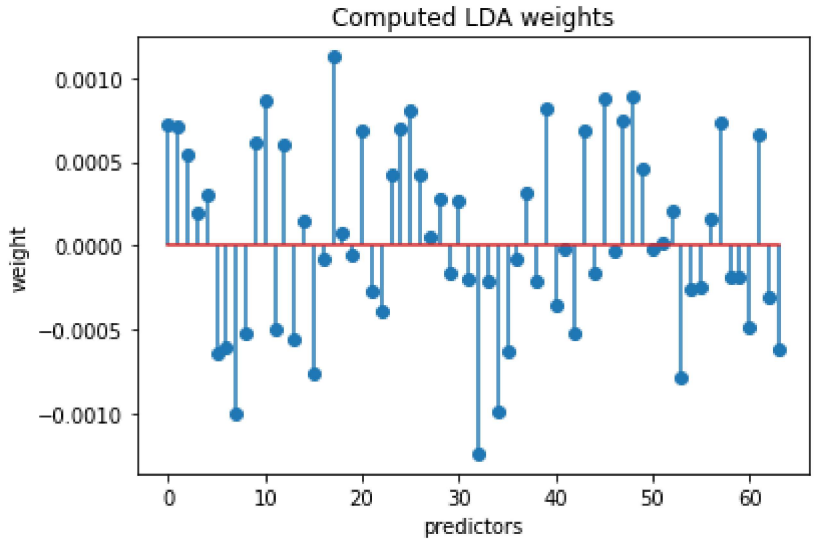
** Task 2.1 ***

1. Implement a function `compute_lda_weights(x, y)` manually, which carries out LDA using the data `x,y`.
2. Apply this function on your training data.
3. Visualize the obtained weight vector $\vec{w}$ using a `plt.stemplot`.

```
#normalize and split data
# apply functions
X_all_norm = preprocess(X_all)

X_train, y_train, X_test, y_test = split_data(X_all_norm,t_all,frac = 0.3, seed = 2903)


# create the lda weights function
def compute_lda_weights(x,y):

    # assign each observation to one class
    X_1 = x[y == True,]
    X_2 = x[y == False,]
```

```
    # compute means within each class
    m_1 = np.mean(X_1,axis = 0)
    m_2 = np.mean(X_2,axis = 0)

    # compute the variances
    def get_s(X,m):
        s = 0
        for jj in range(0,X.shape[0]):
            aux = X[jj] - m
            s = s + np.matmul(aux,aux.T)
        return s
    s_1 = get_s(X_1,m_1)
    s_2 = get_s(X_2,m_2)
    S = s_1 + s_2

    # compute weights
    w = 1/S * (m_2 - m_1)

    return m_1, m_2, w



# X, X_test, t, t_test = split_data(X_all, t_all, seed=1)
m_0, m_1, w_lda = compute_lda_weights(X_train,y_train)

plt.stem(w_lda.flatten(), use_line_collection=True)
plt.title('Computed LDA weights')
plt.ylabel('weight')
plt.xlabel('predictors')
```

```
Text(0.5, 0, 'predictors')
```




```
#the predicted responses:
np.matmul(X_train,w_lda)
```

```
array([-0.00789255,  0.02058851, -0.01211183, -0.01323636, -0.01221882,
       -0.01897203,  0.0134052 , -0.01758203, -0.01188587,  0.00838874,
       -0.0124049 , -0.0126529 ,  0.00692636,  0.00969047, -0.00482659,
        0.01294728, -0.01699111, -0.01356572, -0.01513106,  0.0042109 ,
```

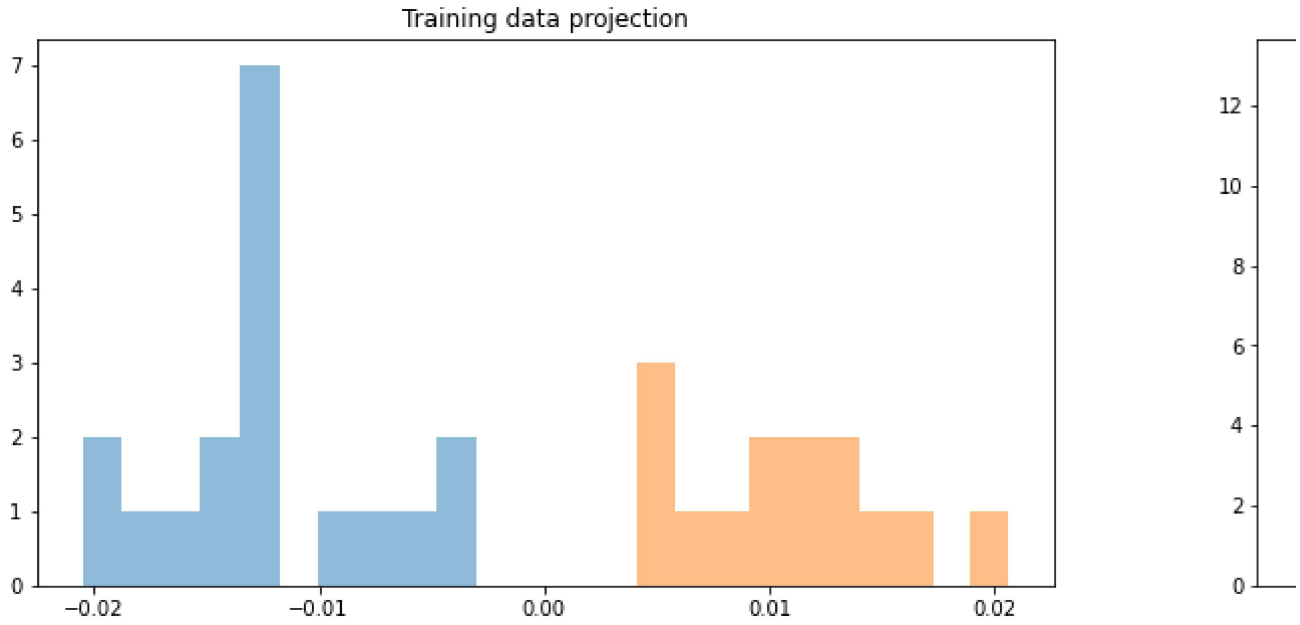
```
-0.00338936,  0.00476007,  0.01107552, -0.02050542, -0.00302484,
 0.00413909,  0.01723565,  0.0151293 , -0.01199066, -0.00960473,
 0.00979522,  0.01209441])
```

Task 2.3

Project the training data and the test data on $\vec{w}$. Visualize the class separation using a two-color histogram.

- Is the class separation good?
- Is there a big difference between training and test data?

```
a,b = np.matmul(X_train[y_train],w_lda), np.matmul(X_train[~y_train],w_lda)
c,d = np.matmul(X_test[y_test],w_lda), np.matmul(X_test[~y_test],w_lda)
fig, ax = plt.subplots(1,2,figsize = (20,5))

ax[0].hist(a ,label='class 0',alpha=0.5)
ax[0].hist(b ,label='class 1',alpha=0.5)
ax[0].set_title("Training data projection")

ax[1].hist(c ,label='class 0',alpha=0.5)
ax[1].hist(d ,label='class 1',alpha=0.5)
ax[1].set_title("Test data projection")

plt.show()
```



Yes, the classes are well separated. The training data has no overlap, while for the test data, there is some overlap around the value of c (~0)

Task 2.4

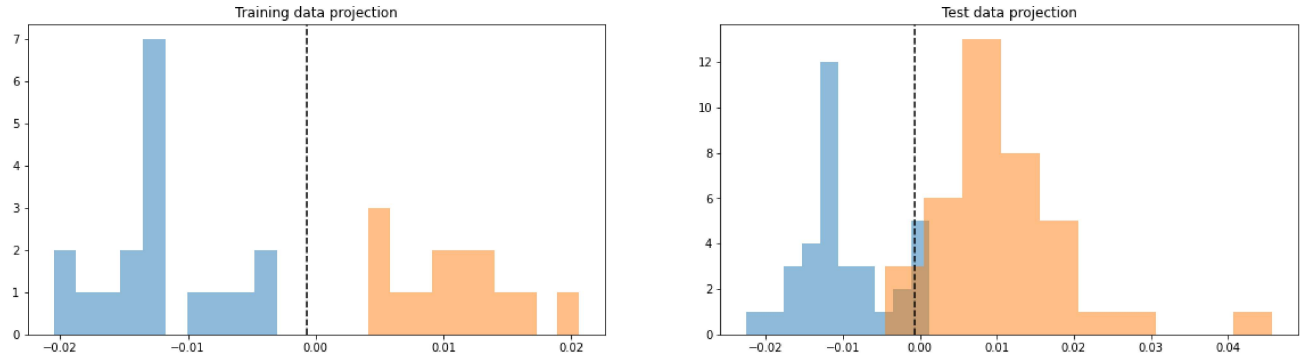
Now we make class predictions based on the projections. Read https://en.wikipedia.org/wiki/Linear_discriminant_analysis#Fisher's_linear_discriminant and compute threshold c for the projected values p based on the training data. Print the value of c and plot c into the histograms of projected values you made before!

Use c to assign class labels for training and test set. Determine the classification errors (in terms of accuracy) on both datasets and print them.

```
c_ = np.matmul(w_lda.T, 0.5 * (m_0 + m_1))
print("threshold c =", c_)
```

```
threshold c = -0.0006842338948584802
```

```
a,b = np.matmul(X_train[y_train],w_lda), np.matmul(X_train[~y_train],w_lda)
c,d = np.matmul(X_test[y_test],w_lda), np.matmul(X_test[~y_test],w_lda)
fig, ax = plt.subplots(1,2,figsize = (20,5))

ax[0].hist(a ,label='class 0',alpha=0.5)
ax[0].hist(b ,label='class 1',alpha=0.5)
ax[0].set_title("Training data projection")
ax[0].axvline(x = c_, color = "black", linestyle = "dashed")

ax[1].hist(c ,label='class 0',alpha=0.5)
ax[1].hist(d ,label='class 1',alpha=0.5)
ax[1].set_title("Test data projection")
ax[1].axvline(x = c_, color = "black", linestyle = "dashed")

plt.show()
```




```
# Determine accuracy of discriminant on training and test set:
pred_train = np.matmul(X_train ,w_lda) <= c_
pred_test = np.matmul(X_test,w_lda) <= c_
correct_train = pred_train == y_train
```

```
correct_test = pred_test == y_test


print("The accuracy on the training set is", str(np.mean(correct_train)), "and on the test
```

```
The accuracy on the training set is 1.0 and on the test it is 0.9452054794520548
```

Task 3: Logistic Regression

Next, we will consider classification using Logistic Regression.

For this task, we will use a different dataset:
It consists of activations from a convolutional neural network (ResNet18) for images of cats and dogs. The dataset contains 2,000 samples (i.e. CNN activations) and 256 features (i.e. the CNN activations have 256 dimensions). A target value of 0 indicates a cat, 1 a dog.

Below, you find all imports that are necessary.

```
#different version of preprocess and split_data because they don't like each other:
def split_data(X, y, frac=0.3, seed=None):
    if seed is not None:
        np.random.seed(seed)

    ### implement the function here
    shuffle_ids = np.array(range(len(y)))
    np.random.shuffle(shuffle_ids)

    slice_ind = int((1-frac) * len(y))

    X = pd.DataFrame(X)

    X = pd.DataFrame.reset_index((X.loc[shuffle_ids,:]),
                                 drop = True)
    y = y[shuffle_ids]

    X_train = X.loc[:slice_ind-1,]
    y_train = y[:slice_ind]

    X_test = X.loc[slice_ind:,]
    y_test = y[slice_ind:]


    return np.array(X_train), np.array(X_test), y_train, y_test

def preprocess(X):

  normdat = deepcopy(X)

  for k in range(X.shape[1]):
    mu = np.mean(normdat[:,k])
    std = np.std(normdat[:,k])

    normdat[:,k] = (normdat[:,k] - mu) / std
```

```
  return normdat


import numpy as np
from sklearn.linear_model import LogisticRegression
import pickle

X_all, t_all = pickle.load(open('data/cnn_features.pickle', 'rb'))
```

Task 3.0: Normalize the data

Make sure the data has has zero mean and variance 1 per feature.

```
X_norm = preprocess(X_all)
X_train, X_test, t_train, t_test = split_data(X_all, t_all)
```

Task 3.1: Iterative Reweighted Least Squares

1. Implement the IRLS algorithm and output at each iteration the current training accuracy. Remember the weight are updated according to:
   $$w' = w - (\Phi^T R \Phi)^{-1} \Phi^T (y - t)$$
   Where y is the prediction, t the ground truth target, R the weighting matrix and Φ the design matrix.
   Hints:
   (a) There is a bias term in logistic regression
   (b) Use a small value for weight init to avoid numerical problems.
2. Apply the IRLS algorithm on data and compute the test accuracy.
3. Compare the results of your implementation to the sklearn implementation of `LogisticRegression(penalty='none')`. Do you get the same result?

```
#function for computing the logistic sigmoid
def logsig(y):
  tmp = 1 + np.exp(-y)
  return 1/tmp


def accuracy(y_true, y_pred):
  n_c = np.sum((y_true == y_pred)) #number of correctly classified
  acc = n_c/len(y_true)
  return acc


def iwls(X, t, abstol = 0.0001, init = 0.1, maxiter = 100):

  X = np.concatenate((np.ones((X.shape[0],1)), X), axis = 1)

  #initialize weights
  weights = np.ones(X.shape[1]) * init
```

```
    #lkh = -np.Inf

    #initialize first mse as infinity
    mse = np.Inf

    #starting guess
    preds = logsig(np.matmul(X, weights))


    for i in range(maxiter):
      mse_old = mse

      #formula components
      R = np.diag(preds * (1-preds))
      inver = np.linalg.inv(np.matmul(X.T, np.matmul(R, X)))
      Hmat = np.matmul(inver, X.T)
      resid = preds - t

      #update weights
      weights = weights -  np.matmul(Hmat, resid)

      #new guess and MSE
      preds = logsig(np.matmul(X, weights))
      mse = np.mean((t-preds)**2)

      #lkh_new = np.sum(t * np.log(preds) + (1-t) * np.log(1-preds))

      print(mse)

      if np.abs(mse - mse_old) < abstol:
        break

    preds = np.array([int(p) for p in preds])
    return preds, weights


  def predict(X_test, w_lr):
    X_test = np.concatenate((np.ones((X_test.shape[0],1)), X_test),
                            axis = 1)
    tmp = np.matmul(X_test, w_lr)
    preds_test = logsig(tmp)

    preds_test = np.array([int(p) for p in preds_test])
    return preds_test


  preds, w_lr = iwls(X_train, t_train)
```

```
    0.05911739215826059
    0.03919275518187618
    0.029369095812085595
    0.021477299976642793
    0.01409904884725693
    0.006224960245967491
    0.0012844081236013233
    0.00018033631467645367
```

```
2.4310698919192303e-05
3.280199744606529e-06
```

```
print("Our Accuracy on the training set is", accuracy(t_train, preds))
```

```
Our Accuracy on the training set is 0.8128571428571428
```

```
preds_test = predict(X_test, w_lr)
print("Our Accuracy on the test set is", accuracy(t_test, preds_test))
```

```
Our Accuracy on the test set is 0.8283333333333334
```

Now, on to sklearn:

```
skmodel = skl.LogisticRegression(penalty = 'none').fit(X_train, t_train)
sk_preds_train = skmodel.predict(X_train)
sk_preds_test = skmodel.predict(X_test)
sk_weights = skmodel.coef_
```

```
print("Our Accuracy on the training set is", accuracy(t_train, sk_preds_train))
```

```
Our Accuracy on the training set is 1.0
```

```
print("Our Accuracy on the test set is", accuracy(t_test, sk_preds_test))
```

```
Our Accuracy on the test set is 0.875
```

Why are these so much higher.... :((((((

Task 3.2: Logistic Regression with Regularization

1. Set sklearn's penalty parameter to `l1` and `l2`. Use the range `np.linspace(0.02, 1, 25)` for the parameter `C`, which controls the strength of regularization. Where is the regularization strongest, for small or big `C`?
   Hint: For `l1` regularization you can use the `saga` solver.
2. Plot strength of regularization over accuracy. Does regularization improve the scores?
3. Visualize the coefficients (or just a subset of all coefficient for a better overview) of the regularized settings and the unregularized setting. What do you observe?
4. Compare the coefficients to the LDA weights.

1. Answer: Regularization is strongest for smaller values of c.
2. Answer: Yes, it does improve the accuracies. For moderate regularization, we get higher accuracies.

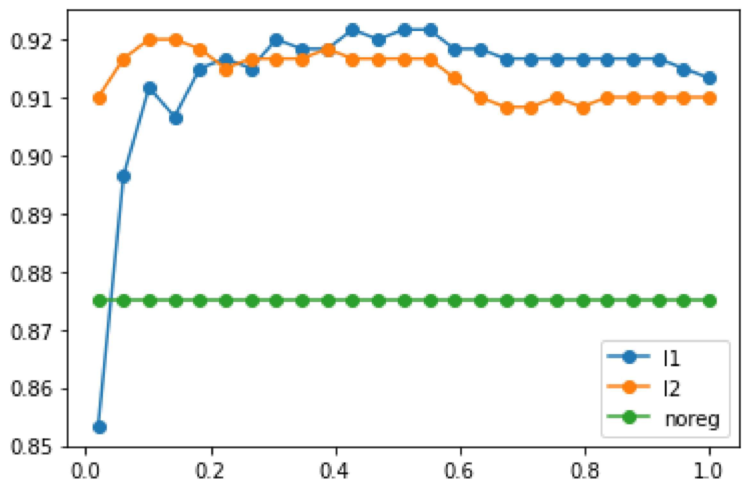
3. Answer: We observe that the coefficients are a lot higher in value for the unregularized

```
cvals = np.linspace(0.02, 1, 25)

accuracies_l1 = []
accuracies_l2 = []
accuracies_nr = []

k = 0
savek = 12

for cval in cvals:
  skmodel_l1 = skl.LogisticRegression(penalty = 'l1', C = cval, solver = 'saga').fit(X_tra
  test_preds_l1 = skmodel_l1.predict(X_test)
  acc_l1 = accuracy(t_test, test_preds_l1)
  accuracies_l1.append(acc_l1)


  skmodel_l2 = skl.LogisticRegression(penalty = 'l2', C = cval).fit(X_train, t_train)
  test_preds_l2 = skmodel_l2.predict(X_test)
  acc_l2 = accuracy(t_test, test_preds_l2)
  accuracies_l2.append(acc_l2)

  if (k == savek):
    l1_weights = skmodel_l1.coef_
    l2_weights = skmodel_l2.coef_

  k += 1
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
  The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
  The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
  The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
  The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
  The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
  The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
  The max_iter was reached which means the coef_ did not converge
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
The max_iter was reached which means the coef_ did not converge
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:330: Convergen
```

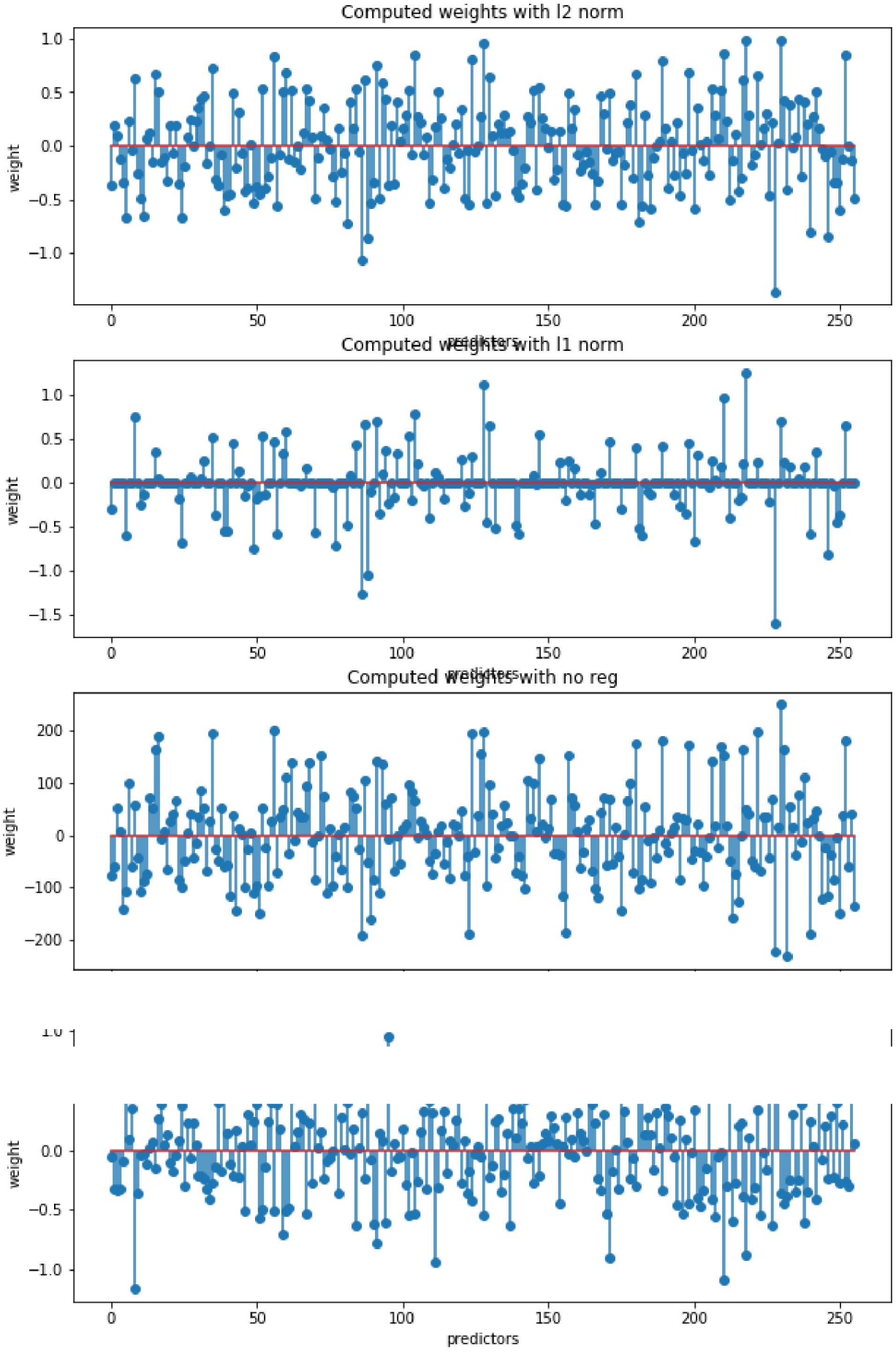
```
regplot = [None, None, None]
acc_nr = accuracy(t_test, sk_preds_test)
l1_plot, = plt.plot(np.linspace(0.02, 1, 25), accuracies_l1, '-o')
l1_plot.set_label('l1')
l2_plot, = plt.plot(np.linspace(0.02, 1, 25), accuracies_l2, '-o')
l2_plot.set_label('l2')
noreg_plot, = plt.plot(np.linspace(0.02, 1, 25), [acc_nr]*25, '-o')
noreg_plot.set_label('noreg')
plt.legend()
plt.show()
```



Plot weights

```
##############LDA results
t_train_bool = np.array([bool(t) for t in t_train])
_m0, _m1, _mdiff, _w_lda = compute_lda_weights(X_train, t_train_bool)

#get accuracy of LDA
####TODO (add offset and such. Can't be bothered right now)
#preds_test = predict(X_test, _w_lda)
#print("Our Accuracy on the test set is", accuracy(t_test, preds_test))


plt.figure(figsize=(10, 16))

plt.subplot(4, 1, 1)
plt.stem(l2_weights.squeeze(), use_line_collection=True)
plt.title('Computed weights with l2 norm')
plt.ylabel('weight')
plt.xlabel('predictors')

plt.subplot(4, 1, 2)
plt.stem(l1_weights.squeeze(), use_line_collection=True)
plt.title('Computed weights with l1 norm')
plt.ylabel('weight')
plt.xlabel('predictors')

plt.subplot(4, 1, 3)
plt.stem(sk_weights.squeeze(), use_line_collection=True)
plt.title('Computed weights with no reg')
plt.ylabel('weight')
plt.xlabel('predictors')

plt.subplot(4, 1, 4)
plt.stem(_w_lda, use_line_collection=True)
plt.title('Computed weights for LDA')
plt.ylabel('weight')
plt.xlabel('predictors')
```

```
Text(0.5, 0, 'predictors')
```



0 s Abgeschlossen um 20:34