Assignment 1

Lukas Jordbru

September 2024

Link to github repos `https://github.com/loshandos99/Fys-2021/tree/main`

Problem 1

1a)

I start with importing the necessary packages for completing this assignment. I then use pandas to load the csv file and to read the amount of songs and features

```
#Importing packages
import pandas as p #pandas to read csv file
import numpy as n #for creating arrays and handling math
import matplotlib.pyplot as plt #to plot
import sklearn.metrics as met

#Reading the csv file into a dataframe
DF = p.read\_csv("SpotifyFeatures.csv",  index\_col=0)


#Getting the dimensions of the data
print(DF.shape)
```

I then get that there is 18 song features and 232 725 songs in the csv file

1b)

Since we are only interrested in the classical and pop songs with the features "liveness" and "loudness", so I am creating two new dataframes. One for pop, and one for classical.

```
#Retrieving the pop and classical part of the dataframes in their own dataframe
Pop_DF = DF.loc["Pop"][["liveness", "loudness"]]
Classical_DF = DF.loc["Classical"][["liveness", "loudness"]]
```

Here the index will be classical for classical songs, and pop for pop songs

1c)

In order too split the dataset 80/20 per class, I will need to find the amount of songs per class. I then combined them into one training dataframe, and one test dataframe. Then can i convert them to the wanted arrays

```
#Finding how many songs are in each class
Pop_songs_length = Pop_DF.shape[0] #-> 9386 songs
Classical_songs_length = Classical_DF.shape[0] #-> 9256 song

#creating training and testing sets with 80/20 split per class
training_DF = p.concat([Pop_DF.iloc[:int(Pop_songs_length*0.8)], Classical_DF.iloc[:int(Classical_songs
test_DF = p.concat([Pop_DF.iloc[int(Pop_songs_length*0.8):], Classical_DF.iloc[int(Classical_songs_leng

#making the matrixes
```

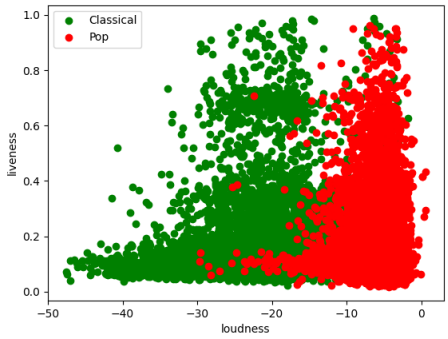

Figure 1: Classical and Pop songs visualized by loudness vs Liveness scales

```
training_matrix = training_DF.to_numpy()
test_matrix = test_DF.to_numpy()

#making the genres vectors
training_genres = training_DF.index.to_numpy()
test_genres = test_DF.index.to_numpy()
```

1d)

Here we can see that it is not quite a gaussian distribution. We have some overlap between the different classes when the loudness is high. This could make it harder for the machine to differentiate between the classes.

Problem 2

2a)

To implement my own logistic discrimination classifier, I am going to start with the sigmoid function. This function is what will predict what class of music each song is. It is going to return a value between 0 and 1, which will stand for the likely hood of that song beeing to class 1. If it returns more than 0.5 we can say with more than 50% likely hood that the song belongs to class 1, so we put it there. Otherwise we put it in the other class. The sigmoid function is defined as

$$f(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} \tag{1}$$

In python this will be

```
    #logistic function
def sigmoid(x):
    """
    The function for returning the predicted value between 0 and 1 to
    identify the class fo the song
    """
    return n.exp(x)/(1+n.exp(x))
```

Next we need to predict it with the right features and weigths in order to get the right predicted value. This we do by using the sigmoid function with the dot product of the features and weight. In order for this to make sense, the features and weights need to be the same length. Since we are using two features, the weigth vector also needs to bee a 2 size vector

```
#Prediction function
def predict(features, weigths):
    """
    features = (x songs, 2 features)
    weigths = (2, 1)
    return => (x songs, 1 value between 0 and 1)

    function for returning the predicted value between 0 and 1 given the
    songs feature and the weitghs.
    This is to determen the given class for each song
    """
    return sigmoid(n.dot(features, weigths))
```

We also need to implement the gradient descent function, which will update the weitghs each epoch based on the predictions and the learning rate. But we also want to show the error as a function of each epoch, so we will need to implement the loss function first. The formula for the loss is

$$L = -\Sigma_i \left(y_i log(f(a^T x_i)) + (1 - y_i) log(1 - f(a^T x_i)) \right) \tag{2}$$

And implemented in python it will look like

```
def cost_function(features, weigths, classification):
    """
    features = (x songs, 2 features)
    weigths = (2, 1)
    classification = (x songs, 1 value)
    return => float

    Function for calculating the error of the function. We want this to be as low as possible
    """
    pred = predict(features, weigths)
    tmp = n.sum(-classification*n.log(pred) + (1-classification)*n.log(1-pred))
    return tmp / len(classification)
```

Then we can also implement the gradient descend function

```
def gradient_descend(features, weigths, classification, learning_rate, epochs):
    """
    features = (x songs, 2 features)
    weigths = (2, 1)
    classification = (x songs, 1 value)
    learning_rate = float
    epochs = int

    return => (weigths (2, 1), cost history list(floats))

    function for calculating the weigths and returning the cost history, that is what the error value i
    """
    cost_history = []
    for _ in range(epochs):

        pred = predict(features, weigths)

        weigths -= n.dot(features.T, pred-classification)*learning_rate/len(classification)

        cost_history.append(cost_function(features, weigths, classification))


    return weigths, cost_history
```

Now we only need to create different learning rates, and starting weigths to run through the training sets

```
#Creating learning rates values, and starting weigth values, and a epoch nr
learnings = [0.005, 0.01, 0.05]
weigth = [0, 0]
epochs = 10000
```

```
#Running through the training set with each learning rate
res = [gradient_descend(training_matrix, weigth, training_genres, learn, epochs) for learn in learnings]


#Plotting the error of each epoch for each learning rate

fig, ax = plt.subplots()
x = n.arange(epochs)
for i in range(len(res)):
    ax.plot(x, res[i][1], label=f"Learning rate {learnings[i]}")
ax.set_ylabel("Cost")
ax.set_xlabel("Epoch")
ax.legend()
plt.show()
```

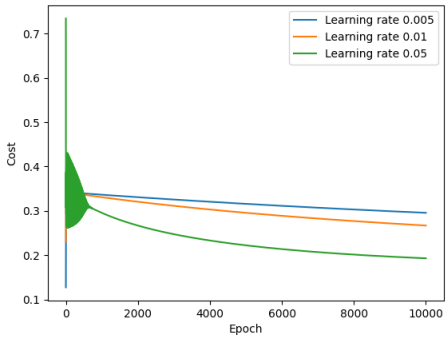

Figure 2: Error as a function of epochs

Here we can see the error as a function of epoch, and we can see that the learning rate 0.05 have the lowest error after 10 000 epochs, which indicates that this will have the highest accuracy of the learning rates. In order to see the accuracy of the different learning rates we will need to make a function to classify after we have predicted what class each song will belong to. And then run a loop through the different learning rates

```
def classify(results, features, classification):
    """
    Results = List[return of gradient descent fucntion, (weigth, cost history)]
    features = features (2, x songs)
    classification = values of 0 or 1 depending on pop or classical music

    return => The accuracy of the model given in percent

    Function for calculating the accuracy of the classification model
    """
    prob = sigmoid(n.dot(features, results[0]))
    prob = n.where(prob >= 0.5, 1, 0)
    print(n.sum(prob))
    accuracy = n.mean(classification == prob)

    return accuracy*100


#Finding the accuracy for the different learning rates
for i in range(len(learnings)):

    print(f"The accuracy for learning rate {learnings[i]} is
    {classify(res[i], training_matrix, training_genres):.2f}")
```

We then get that the accuracy for the learning rates are; 57.25% for 0.005, 61.99% for 0.01, 69.06% for 0.05. We here see that it is indeed highest accuracy for learning rate 0.05.

2b)

To run through the test sets, we only need to run through the gradient descend function one more time with the learned weights. We can do this and get the accuracy with the following code

```
#Finding the accuracy for the test set for each learning rates
for i in range(len(learnings)):
    test_result = gradient_descend(test_matrix, res[i][0], test_genres, learnings[i], 1)
    print(f"The accuracy for the test results with learning rate {learnings[i]}
    is {classify(test_result, test_matrix, test_genres):.2f}")
```

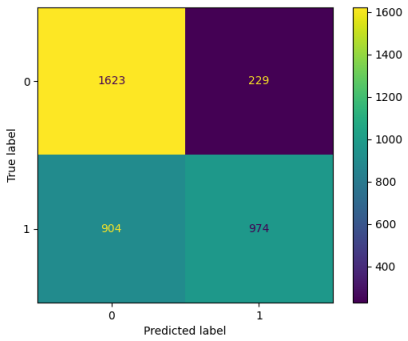

Figure 3: Confusion matrix for the test set

We then get that the accuracy are 56.43 for learning rate 0.005, 61.66 for learning rate 0.01, and 69.62 for learning rate 0.05. Here we can see that the learning rate 0.05 still has the highest accuracys and they are similar to the ones we got for the test set. Still they are not exactly the same, and that is because we are not checking exactly the same songs, but different ones. And they can be some that are harder to identify for the machine, or easier. So there will be some changes, but since we have a farily large sample set we see that it is not that much of a difference.

Problem 3

3a)

To create the confusion matrix I will use the module sklearn. The code to get the confusion matrix plot is given here

```
#Creating the confusion matrixes
pred = sigmoid(n.dot(test_matrix, res[2][0]))
pred = n.where(pred >= 0.5, 1, 0)

conf_mat = met.confusion_matrix(test_genres, pred)
dis = met.ConfusionMatrixDisplay(conf_mat)
dis.plot()
plt.show()
```

We then get the following plot Here the true label is on the upwards scale, and the predicted scale is one the lower scale. The 0 indicates classical, and 1 indicates Pop music. So we can see from the conufsion matrix that the method identified 1623 classical song that actually was classical music, or true negative. Similar we can see 229 false positive, 904 false negative and 974 true positives. This inicates that it struggles to classify the pop songs of the test set. There can be multiple reasons to this, but the most prominent one I can identify is the order of the songs put into the machine while learning. In both the training and test set all the classical songs go in first, and then the pop songs. If we had reversed it, I think we would have had a opposite result. We would have a lot more false positive than false negatives. Maybe it would have gotten a better accuracy if I had put it in random order. But from the figure in problem 1d we can see that the distribution is not quite gaussian, and we have some overlap which makes it hard for the machine to differentiate between the two classes.

0.1 3b)

The confusion matrix gives me information on what the machine predicted to be and what was actually right. And then also gived information on the true positive, false positive, true negative and false negative. This can help us to understand why the accuracy is what it is, and help us figure out what can be done to make the prediction better