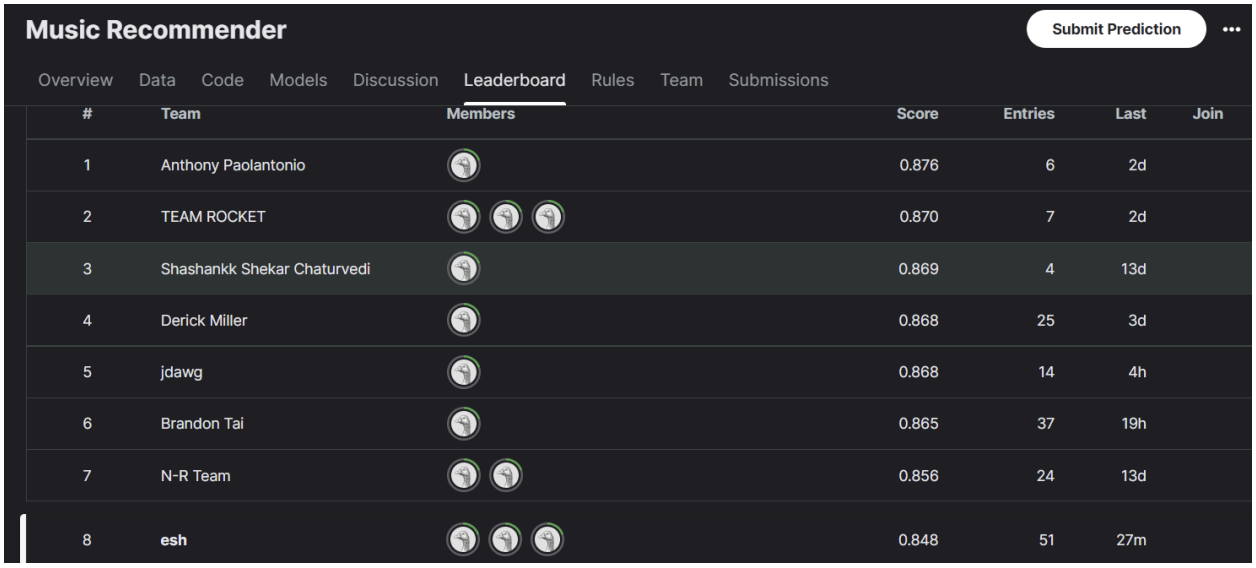
Team Esh: Chris Muro, Andrew Greensweight, Marc DiGeronimo



#	Team	Members	Score	Entries	Last	Join
1	Anthony Paolantonio		0.876	6	2d	
2	TEAM ROCKET		0.870	7	2d	
3	Shashankk Shekar Chaturvedi		0.869	4	13d	
4	Derick Miller		0.868	25	3d	
5	jdawg		0.868	14	4h	
6	Brandon Tai		0.865	37	19h	
7	N-R Team		0.856	24	13d	
8	**esh**		0.848	51	27m	

Current Best Score = .848 Total Successful Submissions = 51

Part 1: For part 1 of the lab the sample code was used to install and configure the environment for pyspark. It was done on google colab because it is computationally expensive. trainItem.data, testItem.data, and output3.csv was uploaded to the colab workspace for use in the code. Output3.csv is our group's previous prediction file that was submitted to kaggle. Trainitem.data was used to train the model and testItem.data was used for predictions by the model. The predictions were written to a csv file. The issue though is that the predictions do not cover the entirety of all the users and tracks.

```
[ ]  1 n_records = training.count()
     2 print(n_records)
     3 testing_count = testing.count()
     4 print(testing_count)
     5 predictions_count = predictions.count()
     6 print(predictions_count)
```

```
12403575
120000
119974
```

Moreover the csv file is not in the correct format for the kaggle submission so both issues needed to be addressed. We read in the predictions.csv file from the ALS model and the old predictions file as pandas dataframes. Next we made an empty userID_list and iterated along the new dataframe to add users in that to the list. Then we can make our 3 1s and 3 0s for each user dataframe. Since the predictions are not between 0-1 we need to select the 3 highest values, put them in a list, sort it and then go through the user data frame again and set the

rating to 1 if it is one of the top 3 scores and 0 if not. Now to get it in the proper format we created a kaggle_dict variable with the keys TrackID and Predictor as required by kaggle submission. We could then iterate through the dataframe and add the correct values to each and then convert that dictionary to a dataframe then to a csv file. Now came the problem of getting the output to the proper amount of rows since some were missing. The first attempt iterated through the new predictions and checked if the value was in the previous prediction. If it was, both indexes were found and the new value of prediction was changed in the old data frame thus updating it.

```
for i in range(len(df_new)):
    if df_new["TrackID"][i] in df_old.values:
        index_old = df_old.loc[df_old["TrackID"] == df_new["TrackID"][i]].index[0]
        index_new = df_new.loc[df_new["TrackID"] == df_new["TrackID"][i]].index[0]
        #df_old.at[index_old, "TrackID"] = df_new.at[index_new, "TrackID"]
        df_old.at[index_old, "Predictor"] = df_new.at[index_new, "Predictor"]
```

The logic was sound but the computation time was not practical. It would take an average of 20 minutes to run the code block on google colab so more efficient ways were attempted. However, this method resulted in the highest score of .766. The next method utilized pandas methods for dataframes.

```
df_update= pd.concat([df_old, df_new]).drop_duplicates(subset='TrackID', keep='last').reset_index(drop=True)
esh_df = df_old[df_old.duplicated("TrackID")]
df_update= pd.concat([df_update, esh_df]).reset_index(drop=True)
```

Here the updated dataframe was the result of the concat method which adds the results of the 2 dataframes and if there are duplicates with the trackID keep the row of the new dataframe and reset index to make it consistent. An intermediate df was used to make sure there were no duplicates in the updated dataframe. This method was much more efficient but resulted in dramatically less accuracy in the .5 and .6 range. Another method was tested using pyspark dataframes. It uses pyspark sql functions to combine 2 dataframes using .join() method to combine the values and if there is overlap give priority to the left dataframe which contains the matrix factorization predictions

```
spark_updated = spark_original.alias('l').join(spark_predict.alias('r'), on='_c0', how='left')\
    .select(
        '_c0',
        f.when(
            ~f.isnull(f.col('r._c1')),
            f.col('r._c1')
        ).otherwise(f.col('l._c1')).alias('value')
    )
```

Now the dataframe can be written into a csv file to be uploaded to kaggle. This method also resulted in very low accuracy but it was fast so it was used for testing. The model was continuously ran and tested using different parameters as discussed in part 2. Many of the tests in this method were in .5 and .6 range. The best results of this bunch was when rank was set to 5 and maxIter was over 20. Ranks lower would result in slightly lower scores and higher ranks were much lower. MaxIter would consistently get better results as it increased up to 20-25 iterations depending on the rank and regParam which was tested using default 0.01 . 0.05, 0.0005 and .1. 0.01 was the best from our testing. When maxIter was over 30 or some

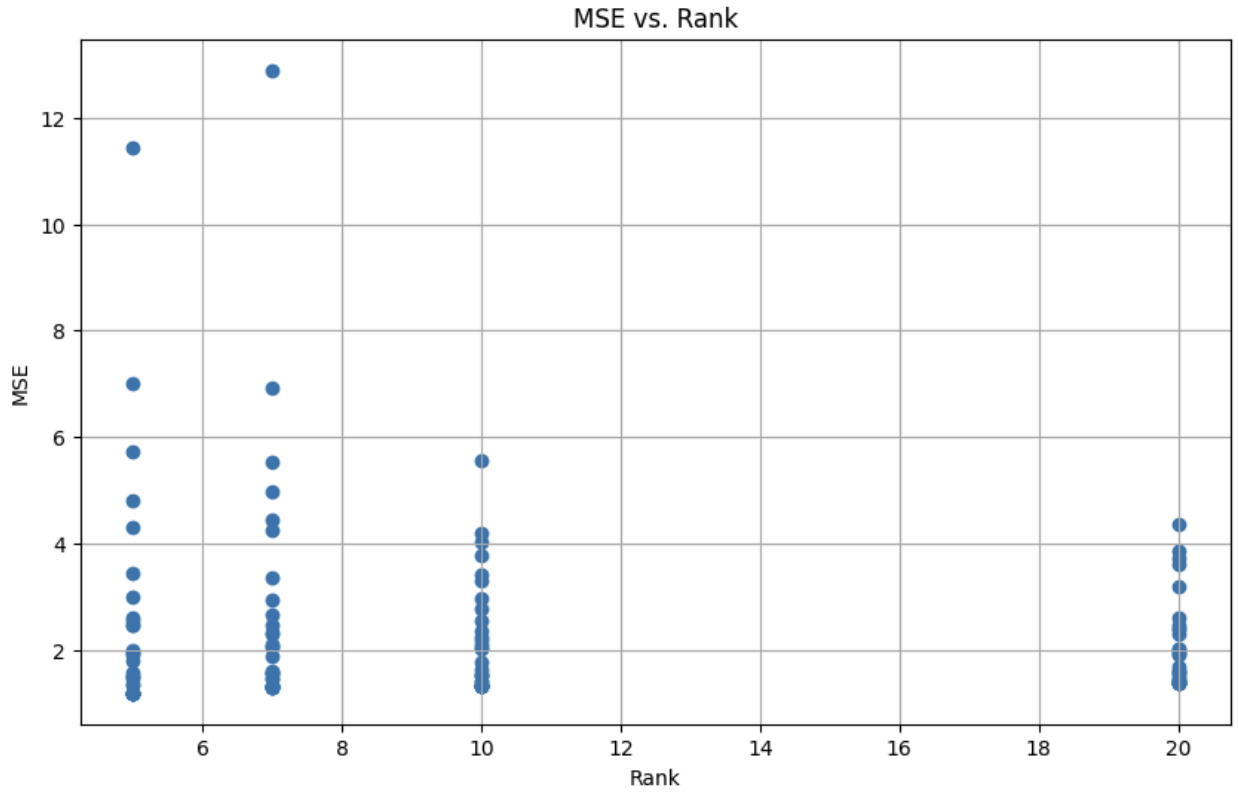
combination of over 20 plus different regParams would receive this error

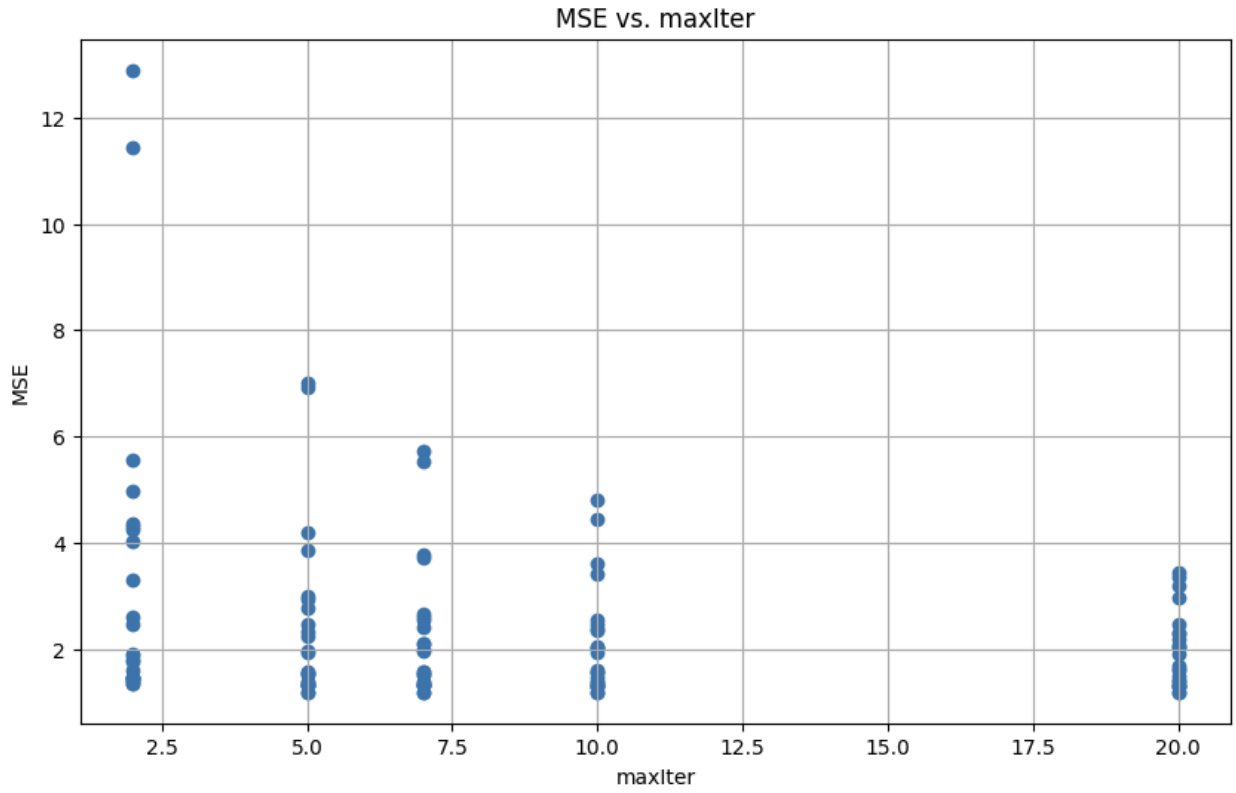
```
    436                 self.socket = self.ssl_context.wrap_socket(
    437                     self.socket, server_hostname=self.java_address)
--> 438             self.socket.connect((self.java_address, self.java_port))
    439             self.stream = self.socket.makefile("rb")
    440             self.is_connected = True

ConnectionRefusedError: [Errno 111] Connection refused
```

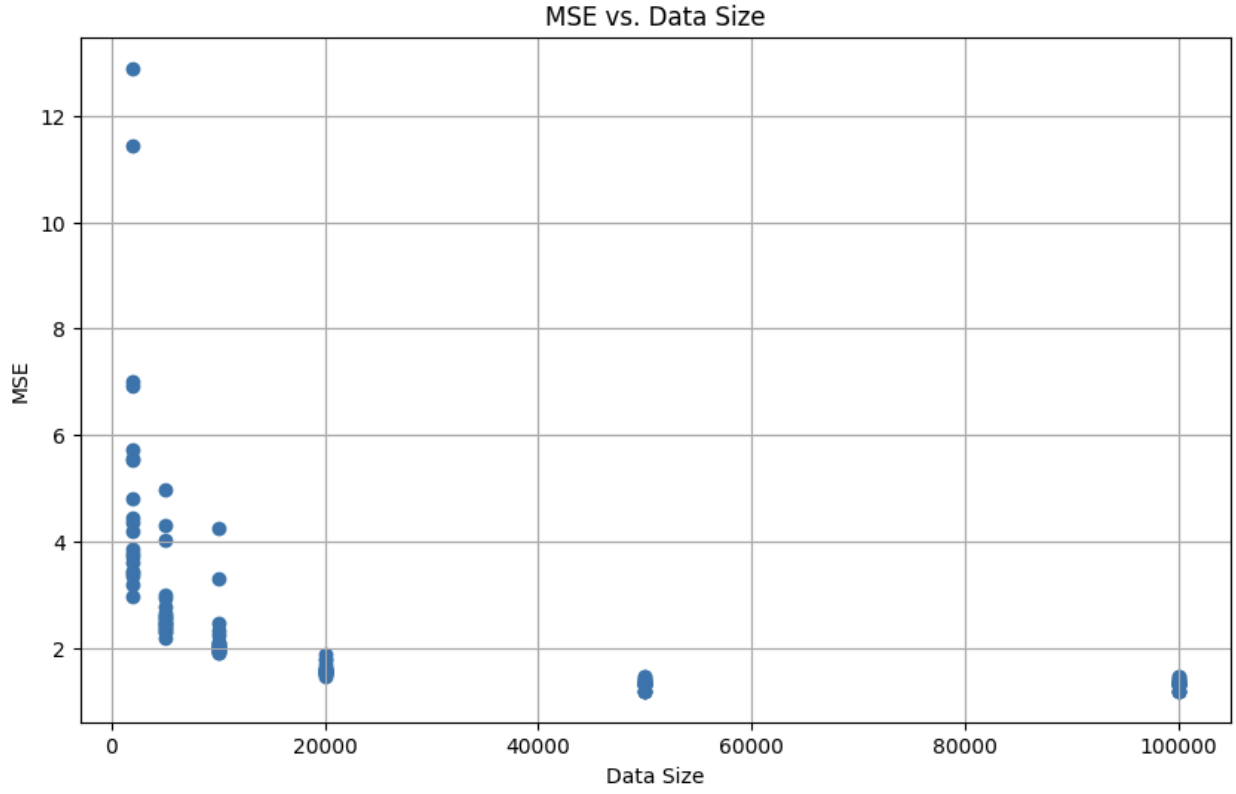
Changing the values back would often fix it after a restart. Cause of such error is unknown to us. Moreover sometimes when running models with same parameters at different times the results would change. This may be due to colab execution blocks and residual memory but we are not sure. Overall none of the combinations were close to our best score of .848 from previous weeks. However this is expected and satisfactory as in upcoming weeks it can be added to ensemble model to increase overall accuracy as it can help in that even with a lower score.

Part 2: For part 2 of the lab, re_u.data file was used. It contains 3 columns of data being the same as the trainItem.data and testItem.data files in the previous part which are in order from left to right, userID, itemID and the rating. First the environment must be configured properly on google colab. This is done by installing the proper programs and seting the enviornment variables and uploading the file. Also re_u.data needs to be split into the train and test set. To do this the dataset was split using np.array_split() function. For 1. maxIter is set to 20 and the rank sizes of 5,7,10 and 20 are tested. With Constant maxIter 20 and rank = 5 the mse was 1.1893121240129518. Rank 7 mse = 1.2858819425771069, Rank = 10, mse = 1.322679774938557, Rank =20 mse = 1.402295299515129. Therefore for these observations we can conclude that at same maxIter and data_size values, as rank increases, mse also increases. This would lead us to using lower rank number for possible models. For 2. rank is set to equal a constant 10 as maxIter takes on the possible values of 2,5,7,10. For maxIter of 2, mse = 1.3911330043443335. MaxIter = 5, mse = 1.315439824961866, MaxIter = 7, mse = 1.3192493847018245. MaxIter = 10, mse = 1.3202044450326698. The results are very interesting. It is assumed that the more iterations the more accurate the model and the less the mse. That is true for this data only to a point however. The lowest mse was at MaxIter 5 and then the mse gradually increased. This may be explained by overfitting as the more iterations the more closley the model will resemble the traning set and loose precision on the test set. This would lead us to choose a lower MaxIter value such as 5 but not too low as 2. However this does not seem to have a drastic impact overall on the mse. For 3, rank is fixed to constant 10, and maxIter is set to constant 20, Here different sizes of data are taken for the size of the train and test set, the values being 2000, 5000, 10000, 20000, 50000 and 100000. For data_size = 2000, mse = 2.9512606448263443. Data_size = 5000, mse = 2.1864230858059064, Data_size = 10000, mse = 2.0322522978298445, Data_size = 20000, mse = 1.6178100700769171. Data_size = 50000, mse = 1.322679774938557. Data_size = 100000, mse = 1.322679774938557. From the results of our data, we can conclude that for the dataset increase in data size is inversely proportional to mse. This would lead us to choose the highest possible data size for our model. Which is consistent with our prediction as more data means more trends and patterns can be identified and averaged out and outliers become less

impactful. From the parameters tested in this HW, we conclude that the data_size will change MSE value most significantly with MaxIter having the least significant impact for our data and models. There are more parameter sets tested with their corresponding mse in the provided .ipynb file for part 2. Highest MSE = 12.89519086944308 occurs with rank = 7 maxIter = 2 and data_size = 2000. Lowest MSE = 1.1884060045633205 occurs with rank = 5, maxIter = 10 and data_size = 100000.




MSE vs. maxIter
MSE
maxIter
2
4
6
8
10
12
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


MSE vs. Data Size
MSE
Data Size
2
4
6
8
10
12
0
20000
40000
60000
80000
100000