Ishuffle Music Recommendation system

In this project we would be implementing a music recommendation system.

Import Packages

```
# Core data manipulation packages
import pandas as pd
import numpy as np

# For our EDA charts
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import seaborn as sns

# For creating sparse matrix
from scipy.sparse import csr_matrix
from scipy.sparse import coo_matrix

# use K-Nearest Neighbors to find cosine distance amongst songs
from sklearn.neighbors import NearestNeighbors
from sklearn.model_selection import train_test_split

# use decomposition in our matrix factorization
import sklearn
from sklearn.decomposition import TruncatedSVD

# set seed for reproducibility of random number initializations
seed = np.random.RandomState(seed=42)
```

Load Data

In this section we read data from all our files

```
#load our files
#training files
triplets_file = 'https://static.turi.com/datasets/millionsong/10000.txt'
songs_metadata_file = 'https://static.turi.com/datasets/millionsong/song_data.csv'

#test file
test_df = pd.read_csv('test_data.csv')

#create first dataframe
song_df_1 = pd.read_table(triplets_file,header=None)
song_df_1.columns = ['user_id', 'song_id', 'listen_count']

#Read song file and create second Dataframe
song_df_2 =  pd.read_csv(songs_metadata_file)
```

```
#Merge the two dataframes above to create input dataframe for recommender systems
train_df = pd.merge(song_df_1, song_df_2.drop_duplicates(['song_id']), on="song_id'
```

Feature Engineering

```
#Merge song title and artist_name columns to make a new feature 'song' which is wha
train_df ['song'] = train_df ['title'].map(str) + " - " + train_df['artist_name']
```

Remove Duplicates from our Dataframe

```
if not train_df[train_df.duplicated(['user_id', 'song'])].empty:
    initial_rows = train_df.shape[0]
    print('Initial dataframe shape {0}'.format(train_df.shape))
    train_df = train_df.drop_duplicates(['user_id', 'song'])
    current_rows = train_df.shape[0]
    print('New dataframe shape {0}'.format(train_df.shape))
    print('Removed {0} rows'.format(initial_rows - current_rows))
```

Exploratory Data Analysis(EDA)

Showing the most popular songs in the dataset

```
print("# Top Songs with most total plays:")
#for all user that have played a song and calculate the total sum of the listen cou
song_rank = (train_df.groupby(['song']).agg({'user_id':'count','listen_count':'sum
print(song_rank.head(20))
```

Visualize Data

We use a barchart to depict the top 20 most listened to songs

```
# our standard bar chart in a function below
def bar_chart_int(x,y,x_label,y_label,title,caption,total_val):
    fig, ax = plt.subplots()
    fig.set_size_inches(16, 5)
    ax = sns.barplot(x[:20], y[:20], palette="PuRd")
    ax.set_xlabel(x_label,fontweight='bold')
    ax.set_ylabel(y_label,fontweight='bold')
    ax.set_title(title,fontweight='bold')
    ax.get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: '{:,}'.fo

    # our bar label placement
    for p in ax.patches:
        height = p.get_height()
        pct = 100*(height/total_val)
        ax.text(p.get_x()+p.get_width()/2.,
                height + 3,
                '{:1.1f}%'.format(pct),
                ha="center",verticalalignment='bottom',color='black')
```

```
    # our caption statement
    ax.text(19, max(y[:20])*0.95, caption, style='italic',fontsize=12,horizontalal:

    plt.xticks(rotation=90)
    plt.show()
```

```
#create a bar chart showing the ranking of top 20 most played songs
c1 = song_rank
x = c1.index
y = c1.userSongPlays
x_label = 'Song Name'
y_label = 'Listen Count'
title = 'Total plays per Song'
caption = 'Percentages are of song plays'
total_val = c1.userSongPlays.sum()
bar_chart_int(x,y,x_label,y_label,title,caption,total_val)
```

Function to get all songs listened to by a user in the dataset

```
def get_user_songs(user):
    user_data = train_df[train_df['user_id'] == user]
    user_items = list(user_data['song'].unique())
    return user_items
#display songs a randomly selected user has listened to in the dataset
index = seed.choice(train_df.shape[0])
uid = train_df.iloc[index]['user_id']
print("User {} has listened to these songs:".format(uid))
for i,song in enumerate(get_user_songs(uid)):
  print("{}. {}".format(i,song))
```

```
def get_actual(user):
    user_data = test_df[test_df['user_id'] == user]
    user_items = list(user_data['songs'].unique())
    return user_items
```

Model Creation

Model 1: Popularity based (Baseline model)

Our first model is the Popularity-based model, this model recommends music to users based on tl

Popularity-based recommender model

```
#Use the popularity based recommender system model to make recommendations
def popularity_recommend(user_id):
      popularity_rank = (train_df.groupby(['song']).agg({'user_id':'count','listen_
      popularity_rank['Rank'] = popularity_rank['popularity_count'].rank(ascending=
      #Get the top 10 recommendations
```

```
    popularity_recommendations = popularity_rank.head(10)

    user_recommendations = popularity_recommendations

    #Add user_id column for which the recommendations are being generated
    user_recommendations['user_id'] = user_id
    #Bring user_id column to the front
    cols = user_recommendations.columns.tolist()
    cols = cols[-1:] + cols[:-1]
    user_recommendations = user_recommendations[cols]

    return user_recommendations
```

Use the popularity model to make some predictions

```
print("---------------------------------------------------------------------")
print("Recommendation based on the most popular songs")
print("---------------------------------------------------------------------")
popularity_recommend("4bd88bfb25263a75bbdd467e74018f4ae570e5df")
```

Model 2: K-Nearest Neighbors

As our first iteration of a basic collaborative recommender, we will build a sparse matrix comparin data will then be passed through a latent mapping algorithm, K-nearest neighbors, to determine cc relationships. This will help us determine which songs are most similar as in shortest distance ap

Prepare Sparse Matrix

In this section, we fit data into a sparse matrix of songs (row) vs. user (column). This matrix captu with number of listens in each respective cell.

```
# function to fit dataframe into a sparse matrix of song name (row) vs user (colum
# in terms of listen count
def data_to_sparse(data,index,columns,values):
    pivot = data.pivot(index=index,columns=columns,values=values).fillna(0)
    sparse = csr_matrix(pivot.values)
    print(sparse.shape)
    return pivot,sparse
```

Use K Nearest Neighbors to determine cosine distance amongst songs

```
def fit_knn(sparse):
    knn = NearestNeighbors(metric='cosine')
    knn.fit(sparse)
    print(knn)
    return knn
```

Create Sparse Matrix using our DataFrame

```
pivot_df,sparse_df = data_to_sparse(train_df,index='song',columns='user_id',values=
```

```
pivot_df,sparse_df = data_to_sparse(train_df,index='song',columns='user_id',values=
```

```
(9953, 76353)
```

```
pivot_df.head(10)
```

Fit our sparse matrix to our knn model

```
knn = fit_knn(sparse_df)
```

In this function we make our recommendations based on our knn model, we lookup song similarit
matrix, with cosine distance in parentheses.

```
def knn_recommend(user_id,data,song,model,k):
    distances, indices = (model.kneighbors(data.loc[song].values.reshape(1,-1),n_ne
    predicted =[]
    for i in range(0,len(distances.flatten())):
        if i == 0:
            print(("KNN Song Recommendations for user {} cause they like '{}':\n".
                   format(user_id,song)))
        else:
            print(('{}: {} ({:.3f})'.format(i,data.index[indices.flatten()[i]],dist
            predicted.append(data.index[indices.flatten()[i]])
    return predicted
```

Model 3 Matrix Factorization

Similar with kNN, we convert our training data into a 2D matrix (called a utility matrix here) and fill
user_id and our column is song

```
pivot_df2 = train_df.pivot(index = 'user_id', columns = 'song', values = 'listen_co
pivot_df2.head()
```

We then transpose this utility matrix, so that the songs become rows and user_id become column

```
X = pivot_df2.values.T
X.shape
```

```
(9953, 76353)
```

After using TruncatedSVD to decompose it, we fit it into the model for dimensionality reduction. Tl
columns since we must preserve the song names. We choose n_components = 12 for just 12 late
dimensions have been reduced significantly from 9953 X 76353 to 9953 X 12

```
SVD = TruncatedSVD(n_components=12, random_state=17)
matrix = SVD.fit_transform(X)
matrix.shape
```

We calculate the Pearson’s R correlation coefficient for every song pair in our final matrix.

```
correlation = np.corrcoef(matrix)
correlation.shape
```

We recommend songs using this function. Given a song we find all songs that have high correlatic
them.

```
def matrix_factorization_recommend(user_id, song):
  songs = pivot_df2.columns
  song_list = list(songs)
  song_index = song_list.index(song)
  correlation_song_index  = correlation[song_index]
  print("\nMatrix Factorization recommendation for user {} because they like '{}':'
  for i, song in enumerate(list(songs[(correlation_song_index > 0.989)])):
      print("{}. {}".format(i,song))
  return list(songs[(correlation_song_index > 0.989)])
```

Model Evaluation

Make predictions using our two models, calculate and compare the precision and recall of the twc

```
user_id ="4bd88bfb25263a75bbdd467e74018f4ae570e5df"
user_songs= get_user_songs(user_id)
actual_songs= get_actual(user_id)
#knn recommended
knn_predicted = knn_recommend(user_id, pivot_df,user_songs[5],knn,10)
#Matrix factorization recommended
matrix_fac_predicted = matrix_factorization_recommend(user_id, user_songs[5])
```

We define two functions to calculate the precision and recall for our models.

```
def calc_precision(predicted, actual):
        prec = [value for value in predicted if value in actual]
        prec = np.round(float(len(prec)) / float(len(predicted)), 4) #tp/tp+fp
        return prec

def calc_recall(predicted, actual):
        reca = [value for value in predicted if value in actual]
        reca = np.round(float(len(reca)) / float(len(actual)), 4) #tp/tp+fn
        return reca
```

Calculate Precision and Recall for KNN

```
knn_prec = calc_precision(knn_predicted, actual_songs)
knn_rec = calc_recall(knn_predicted, actual_songs)
```

Calculate Precision and Recall for Matrix Factorization

```
maf_prec = calc_precision(matrix_fac_predicted, actual_songs)
maf_rec = calc_recall(matrix_fac_predicted, actual_songs)
```

Compare Precision and Recall for Both Models

```
if knn_prec > maf_prec:
  print("knn has higher precision of {} compared to the precision of matrix factor:
  print("--------------------------------------------------------------------------
else:
  print("MAF has higher precision of {} compared to the precision of KNN which is ·
  print("--------------------------------------------------------------------------

if knn_rec > maf_rec:
  print("knn has higher recall of {} compared to the recall of matrix factorizatio
  print("--------------------------------------------------------------------------
else:
  print("MAF has higher precision of {} compared to the recall of KNN which is {}".
  print("--------------------------------------------------------------------------
```

Conclusion

After evaluating our models we can see that knn had a higher precision and recall compared to the

Demo

```
#randomly select a user from the dataset
index = seed.choice(train_df.shape[0])
uid = train_df.iloc[index]['user_id']
#get songs userhas listened to
user_songs= get_user_songs(uid)
song_index = seed.choice(len(user_songs))
print("User {} has listened to this song '{}' therefore they will like the followin
#knn recommend similar songs
knn_predicted = knn_recommend(uid, pivot_df,user_songs[song_index],knn,10)
#Matrix factorization recommend similar songs
matrix_fac_predicted = matrix_factorization_recommend(uid, user_songs[song_index])
```