Explanation of the code

Moschopoulos Apostolos

1st of December 2021

Abstract

Spotify Million Playlist Dataset

1 Representing the information

The dataset has been given in 1000 JSON files. Each file contains 1000 playlists and some meta information. In this first section we will meticulously explain step-by-step how the information was extracted from the initial given files and represented using python and pandas utilities.

1.1 Importing the data

```
import json
import numpy as np
import pandas as pd
```

In those lines of code numpy, pandas and the json module are imported. Pandas is a very powerful tool for data analysis so we will store our data into a pandas dataframe.

```
def read_from_json(start):
    stop = start + 999
    path = 'Spotify_Playlists/mpd.slice.' + str(start) + '-' + str(stop) + '.json'
    with open(path) as data_file:
        d = json.load(data_file)
    df = pd.json_normalize(d, 'playlists')
    return df
```

In the second cell, we are defining a function that creates a file path, then reaches a particular JSON file, reads it's input and stores them into a pandas data frame.

```
%%time
#the X in range(X) to determine the number of json we will use
train = pd.concat({i : read_from_json(i) for i in range(0,10000,1000)})
train.shape
```

```
Wall time: 11.3 s

(10000, 12)
```

This loop calls the above determined function and creates the complete data frame, utilizing a pandas method called 'concat' (pandas.concat). **PLEASE NOTE:** Due to the lack of resources, a mere chunk of the whole dataset Spotify is providing us was used. That loop is designed to reach 10 JSON files hence only 10 thousand out of the one million playlists were used. This leads to inevitable drop in success rate metrics, not only because the model has too few instances (playlists) to learn from, but also from the fact that the number of individual tracks the model is going to use in order to complete the playlists from challenge set is going to be *by far* shorter than the real one, which simply means the model doesn't even **know a big percentage of tracks that exist** so it can't possibly include them in the complete playlists.

```
%%time
with open('Spotify_Playlists/challenge_set.json') as data_file:
    d= json.load(data_file)

test = json_normalize(d, 'playlists')
test.shape

train_test_data = pd.concat([train,test])
```

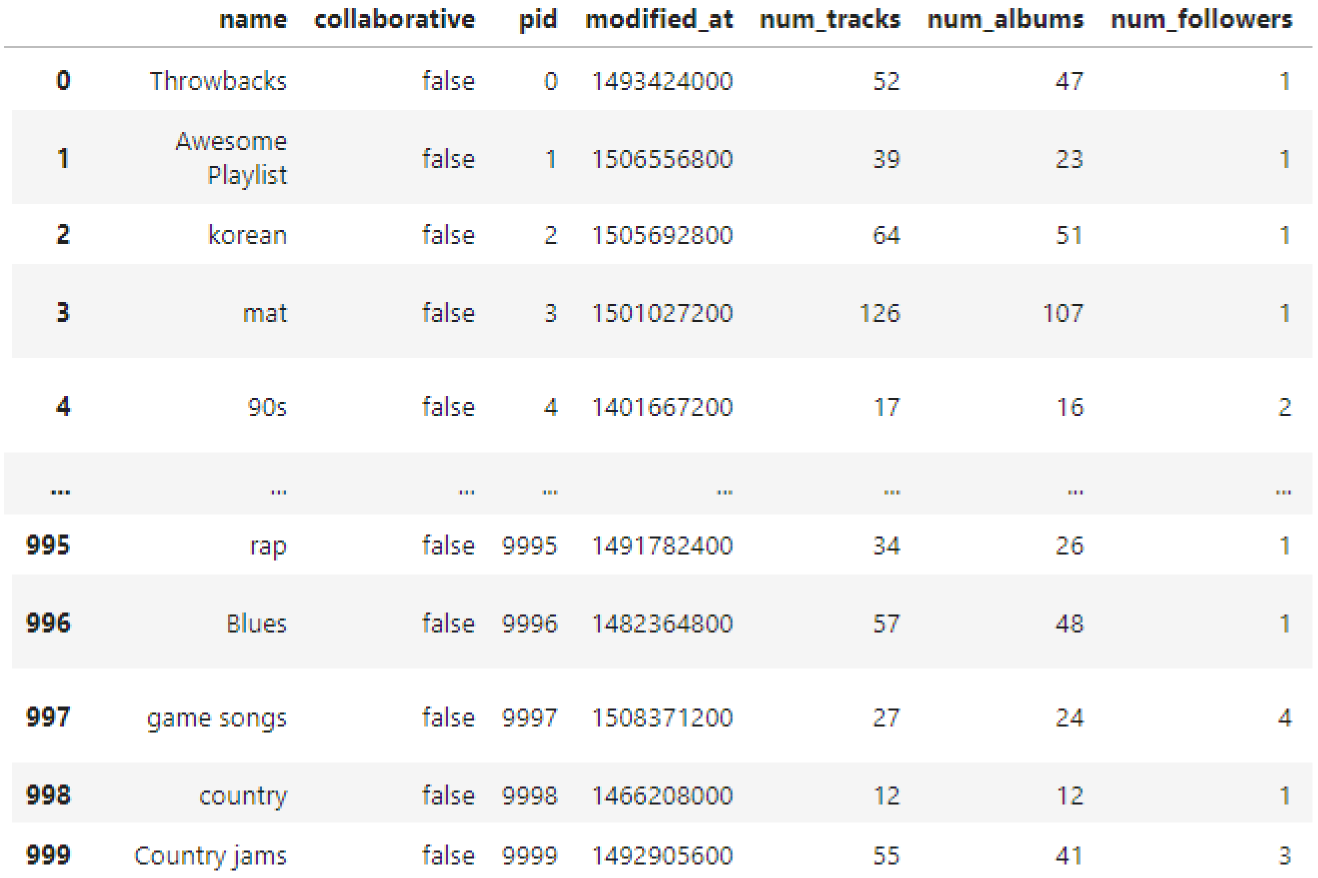
In this cell we put the challenge set, a.k.a the incomplete playlists that need to be filled, into the program. The final line is again concatenating the training and test datasets into a single pandas data frame. Yes, that is required because as soon as we create the sparse matrix (more info about this in section 1.3), we need the challenge set information in order to depict the correlation between tracks and playlists.

We now have established the dataset we will be working with, and no additional info shall be imported. Before we start to work and analyse it we will show it in the initial raw form. The dataframe consists of 1000 rows -each row contains information for one playlist- and 12 columns. Let's look at what each column is offering us:

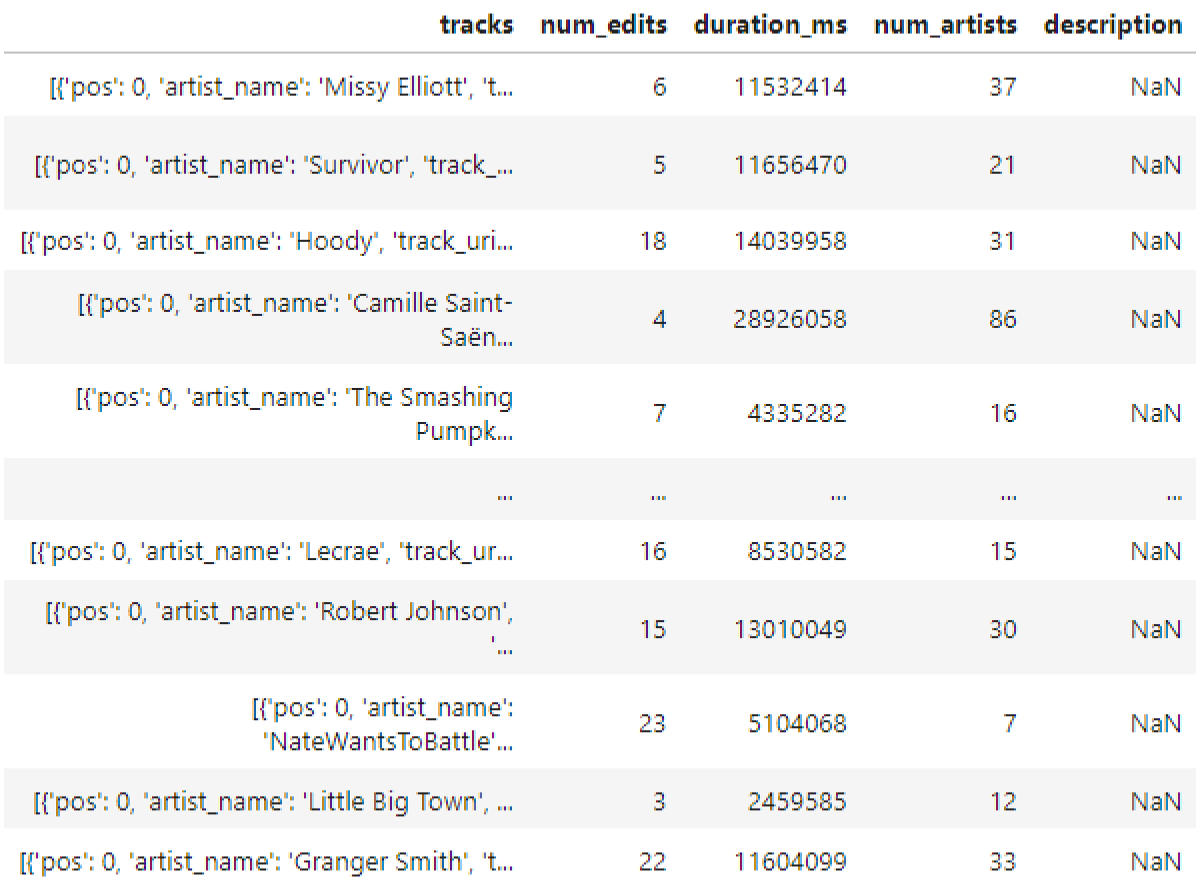
- **name** string - the name of the playlist
- **collaborative** boolean - if true, the playlist is a collaborative playlist. Multiple users may contribute tracks to a collaborative playlist.
- **pid** integer - playlist id - the MPD ID of this playlist. This is an integer between 0 and 999,999. **NOTE THAT** the pid will only be between 0 and 9,999 in this data frame due to the smaller amount of input.
- **modified_at** seconds - timestamp (in seconds since the epoch) when this playlist was last updated. Times are rounded to midnight GMT of the date when the playlist was last updated.
- **num_tracks** the number of tracks in the playlist
- **num_albumns** the number of unique albums for the tracks in the playlist

- **num_followers** the number of followers this playlist had at the time the MPD was created. (Note that the follower count does not including the playlist creator)
- **tracks** an array of information about each track in the playlist. Each element in the array is a dictionary with the some fields we will later talk about
- **num_edits** the number of separate editing sessions. Tracks added in a two hour window are considered to be added in a single editing session.
- **duration_ms** the total duration of all the tracks in the playlist (in milliseconds)
- **num_artists** the total number of unique artists for the tracks in the playlist.
- **description** optional string - if present, the description given to the playlist. Note that user-provided playlist descriptions are a relatively new feature of Spotify, so most playlists do not have descriptions.

You can acquire the information above from *README.md* file. We are producing the shape and some rows of the data frame:

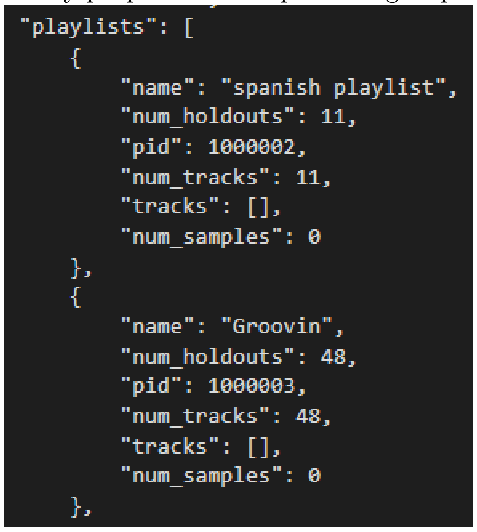


	name	collaborative	pid	modified_at	num_tracks	num_albums	num_followers
0	Throwbacks	false	0	1493424000	52	47	1
1	Awesome Playlist	false	1	1506556800	39	23	1
2	korean	false	2	1505692800	64	51	1
3	mat	false	3	1501027200	126	107	1
4	90s	false	4	1401667200	17	16	2
...	...	...	...	...	...	...	...
995	rap	false	9995	1491782400	34	26	1
996	Blues	false	9996	1482364800	57	48	1
997	game songs	false	9997	1508371200	27	24	4
998	country	false	9998	1466208000	12	12	1
999	Country jams	false	9999	1492905600	55	41	3



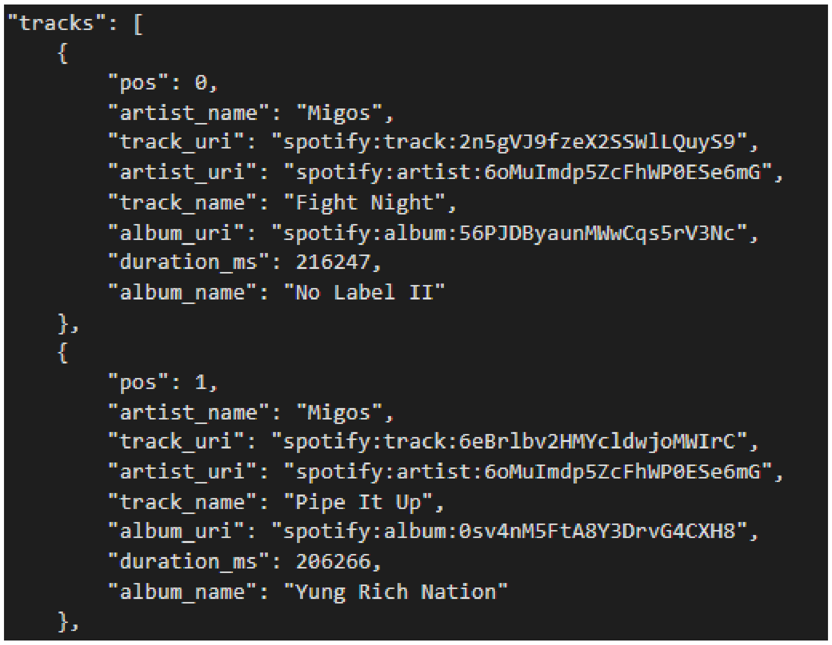
tracks	num_edits	duration_ms	num_artists	description
[{'pos': 0, 'artist_name': 'Missy Elliott', 't...	6	11532414	37	NaN
[{'pos': 0, 'artist_name': 'Survivor', 'track_...	5	11656470	21	NaN
[{'pos': 0, 'artist_name': 'Hoody', 'track_uri...	18	14039958	31	NaN
[{'pos': 0, 'artist_name': 'Camille Saint-Saën...	4	28926058	86	NaN
[{'pos': 0, 'artist_name': 'The Smashing Pumpk...	7	4335282	16	NaN
...	...	...	...	...
[{'pos': 0, 'artist_name': 'Lecrae', 'track_ur...	16	8530582	15	NaN
[{'pos': 0, 'artist_name': 'Robert Johnson', '...	15	13010049	30	NaN
[{'pos': 0, 'artist_name': 'NateWantsToBattle'...	23	5104068	7	NaN
[{'pos': 0, 'artist_name': 'Little Big Town', ...	3	2459585	12	NaN
[{'pos': 0, 'artist_name': 'Granger Smith', 't...	22	11604099	33	NaN

Furthermore, we are also importing the challenge set that has a slightly different form. We use the method DataFrame.from_dict because the data stored in the JSON file are in the form of a nested dictionary. For clarity purposes we are presenting 2 spots from the challenge_set JSON file:



```
"playlists": [
    {
        "name": "spanish playlist",
        "num_holdouts": 11,
        "pid": 1000002,
        "num_tracks": 11,
        "tracks": [],
        "num_samples": 0
    },
    {
        "name": "Groovin",
        "num_holdouts": 48,
        "pid": 1000003,
        "num_tracks": 48,
        "tracks": [],
        "num_samples": 0
    },
```

PLEASE NOTE there are some empty playlists that need to be filled which means those ones need to be more specially treated.


```
"tracks": [
    {
        "pos": 0,
        "artist_name": "Migos",
        "track_uri": "spotify:track:2n5gVJ9fzeX2SSWlLQuyS9",
        "artist_uri": "spotify:artist:6oMuImdp5ZcFhWP0ESe6mG",
        "track_name": "Fight Night",
        "album_uri": "spotify:album:56PJDByaunMWwCqs5rV3Nc",
        "duration_ms": 216247,
        "album_name": "No Label II"
    },
    {
        "pos": 1,
        "artist_name": "Migos",
        "track_uri": "spotify:track:6eBrlbv2HMYcldwjoMWIrC",
        "artist_uri": "spotify:artist:6oMuImdp5ZcFhWP0ESe6mG",
        "track_name": "Pipe It Up",
        "album_uri": "spotify:album:0sv4nM5FtA8Y3DrvG4CXH8",
        "duration_ms": 206266,
        "album_name": "Yung Rich Nation"
    },
```

The columns here are slightly different. Let's check what *README.md* file has to say about the contents of those columns

- **name** (optional) - the name of the playlist. For some challenge playlists, the name will be missing.
- **num_holdouts** the number of tracks that have been omitted from the playlist
- **pid** the playlist ID
- **num_tracks** the total number of tracks in the playlist.
- **tracks** a (possibly empty) array of tracks that are in the playlist.
- **num_samples** the number of tracks included in the playlist

We are receiving them in the form below:

```python
test_df = json.load(open(r'Spotify_Playlists/challenge_set.json','r'))
test_df = pd.DataFrame.from_dict(test_df['playlists'],orient='columns')
```

```python
test_df.drop(test_df[test_df['num_samples'] == 0].index, inplace = True) #Drop records from challenge set
                                                                          #that dont contain any track
```

```python
test_df.head(15)
```

	name	num_holdouts	pid	num_tracks	tracks	num_samples
1000	Party	70	1000000	75	[{'pos': 0, 'artist_name': 'AronChupa', 'track...	5
1001	school	73	1000016	78	[{'pos': 0, 'artist_name': 'Alesso', 'track_ur...	5
1002	Modern Music	63	1000020	68	[{'pos': 0, 'artist_name': 'Banks', 'track_uri...	5
1003	lit 2.0	49	1000023	54	[{'pos': 0, 'artist_name': 'Galantis', 'track_...	5
1004	bbq	71	1000040	76	[{'pos': 0, 'artist_name': 'Stick Figure', 'tr...	5
1005	Classic Jams	60	1000049	65	[{'pos': 0, 'artist_name': 'Bon Jovi', 'track_...	5
1006	dank	75	1000052	80	[{'pos': 0, 'artist_name': 'Tycho', 'track_uri...	5
1007	Dance	57	1000068	62	[{'pos': 0, 'artist_name': 'Fetty Wap', 'track...	5
1008	Pop Playlist	49	1000082	54	[{'pos': 0, 'artist_name': 'Drake', 'track_uri...	5
1009	Oldies	48	1000084	53	[{'pos': 0, 'artist_name': 'The Beach Boys', '...	5
1010	windows down	50	1000092	55	[{'pos': 0, 'artist_name': 'Vincent Mango', 't...	5
1011	august 2017	57	1000094	62	[{'pos': 0, 'artist_name': 'Calvin Harris', 't...	5
1012	clasic rock	53	1000095	58	[{'pos': 0, 'artist_name': '38 Special', 'trac...	5
1013	stubborn love	86	1000100	91	[{'pos': 0, 'artist_name': 'Matt Nathanson', '...	5
1014	Kpop	56	1000104	61	[{'pos': 0, 'artist_name': 'LUNA', 'track_uri'...	5

Even though playlists that don't include any song were dropped out of the dataset, there has been applied a natural language processing method called tf-idf with whom by analysing the empty playlists title, we can perform satisfactory predictions.

1.2 Understanding the form of the dataset

We have observed the dataset. With this current form, tracks are unreachable. We need to extract the information from *'tracks'* column. This can be achieved easily, by using the cat.codes and appending methods . But first, let's organize the data frame a little to lessen confusion regarding the pid column.

```
for i, row in test_df.iterrows(): #change the enumeration of challenge set pid's
    test_df.at[i,'pid'] = int(9000 + i)
```

```
test_df.head(20)
```

	name	num_holdouts	pid	num_tracks	tracks	num_samples
1000	Party	70	10000	75	[{'pos': 0, 'artist_name': 'AronChupa', 'track...	5
1001	school	73	10001	78	[{'pos': 0, 'artist_name': 'Alesso', 'track_ur...	5
1002	Modern Music	63	10002	68	[{'pos': 0, 'artist_name': 'Banks', 'track_uri...	5
1003	lit 2.0	49	10003	54	[{'pos': 0, 'artist_name': 'Galantis', 'track_...	5
1004	bbq	71	10004	76	[{'pos': 0, 'artist_name': 'Stick Figure', 'tr...	5
1005	Classic Jams	60	10005	65	[{'pos': 0, 'artist_name': 'Bon Jovi', 'track_...	5
1006	dank	75	10006	80	[{'pos': 0, 'artist_name': 'Tycho', 'track_uri...	5
1007	Dance	57	10007	62	[{'pos': 0, 'artist_name': 'Fetty Wap', 'track...	5
1008	Pop Playlist	49	10008	54	[{'pos': 0, 'artist_name': 'Drake', 'track_uri...	5
1009	Oldies	48	10009	53	[{'pos': 0, 'artist_name': 'The Beach Boys', '...	5
1010	windows down	50	10010	55	[{'pos': 0, 'artist_name': 'Vincent Mango', 't...	5
1011	august 2017	57	10011	62	[{'pos': 0, 'artist_name': 'Calvin Harris', 't...	5
1012	clasic rock	53	10012	58	[{'pos': 0, 'artist_name': '38 Special', 'trac...	5
1013	stubborn love	86	10013	91	[{'pos': 0, 'artist_name': 'Matt Nathanson', '...	5
1014	Kpop	56	10014	61	[{'pos': 0, 'artist_name': 'LUNA', 'track_uri'...	5
1015	Halloween	52	10015	57	[{'pos': 0, 'artist_name': 'Louis Armstrong', ...	5
1016	Romanticas	47	10016	52	[{'pos': 0, 'artist_name': 'Camila', 'track_ur...	5
1017	relax	50	10017	55	[{'pos': 0, 'artist_name': 'Us The Duo', 'trac...	5
1018	New Songs!	50	10018	55	[{'pos': 0, 'artist_name': 'Timeflies', 'track...	5
1019	its not a phase mom	50	10019	55	[{'pos': 0, 'artist_name': 'Radiohead', 'track...	5

Now pid utility has been established.

```
#itterate every playlist and organise dataframe with trackid as a key
song_playlist_array = []
for index,row in train_test_data.iterrows():
  for track in row['tracks']:
    song_playlist_array.append([track['track_uri'],track['artist_name'],track['track_name'],row['pid'],row['num_holdouts']])
song_playlist_df = pd.DataFrame(song_playlist_array,columns=['trackid', 'artist_name', 'track_name', 'pid', 'num_holdouts'])

print(song_playlist_df.shape)
song_playlist_df.head(10)
```

```
(945712, 5)
```

	trackid	artist_name	track_name	pid	num_holdouts
0	spotify:track:0UaMYEvWZi0ZqiDOoHU3YI	Missy Elliott	Lose Control (feat. Ciara & Fat Man Scoop)	0	NaN
1	spotify:track:6I9VzXrHxO9rA9A5euc8Ak	Britney Spears	Toxic	0	NaN
2	spotify:track:0WqIKmW4BTrj3eJFmnCKMv	Beyoncé	Crazy In Love	0	NaN
3	spotify:track:1AWQoqb9bSvzTjaLralEkT	Justin Timberlake	Rock Your Body	0	NaN
4	spotify:track:1lzr43nnXAijIGYnCT8M8H	Shaggy	It Wasn't Me	0	NaN
5	spotify:track:0XUfyU2QviPAs6bxSpXYG4	Usher	Yeah!	0	NaN
6	spotify:track:68vgtRHr7iZHpzGpon6Jlo	Usher	My Boo	0	NaN
7	spotify:track:3BxWKCI06eQ5Od8TY2JBeA	The Pussycat Dolls	Buttons	0	NaN
8	spotify:track:7H6ev70Weq6DdpZyyTmUXk	Destiny's Child	Say My Name	0	NaN
9	spotify:track:2PpruBYCo4H7WOBJ7Q2EwM	OutKast	Hey Ya! - Radio Mix / Club Mix	0	NaN

In that cell we created a new data frame called 'song_playlist_df'. It's columns consist of the nested features of the column 'tracks'. **Note that** train_test_data is a concatenation of train and test_df.

```
%time
song_playlist_df['track_index'] = song_playlist_df['trackid'].astype('category').cat.codes #creating a number index of
print(len(song_playlist_df['track_index'].unique()))                                         #every track instead of url
song_playlist_df.head(10)
```

```
Wall time: 0 ns
189583
```

	trackid	artist_name	track_name	pid	num_holdouts	track_index
0	spotify:track:0UaMYEvWZi0ZqiDOoHU3YI	Missy Elliott	Lose Control (feat. Ciara & Fat Man Scoop)	0	NaN	12240
1	spotify:track:6I9VzXrHxO9rA9A5euc8Ak	Britney Spears	Toxic	0	NaN	153232
2	spotify:track:0WqIKmW4BTrj3eJFmnCKMv	Beyoncé	Crazy In Love	0	NaN	13145
3	spotify:track:1AWQoqb9bSvzTjaLralEkT	Justin Timberlake	Rock Your Body	0	NaN	28795
4	spotify:track:1lzr43nnXAijIGYnCT8M8H	Shaggy	It Wasn't Me	0	NaN	43407
5	spotify:track:0XUfyU2QviPAs6bxSpXYG4	Usher	Yeah!	0	NaN	13395
6	spotify:track:68vgtRHr7iZHpzGpon6Jlo	Usher	My Boo	0	NaN	149556
7	spotify:track:3BxWKCI06eQ5Od8TY2JBeA	The Pussycat Dolls	Buttons	0	NaN	77841
8	spotify:track:7H6ev70Weq6DdpZyyTmUXk	Destiny's Child	Say My Name	0	NaN	177068
9	spotify:track:2PpruBYCo4H7WOBJ7Q2EwM	OutKast	Hey Ya! - Radio Mix / Club Mix	0	NaN	59026

We are almost finished with manipulating the shape and form of the data frame. All that's left is mapping the data frame as a *sparse matrix*. In the next subsection we are going to explain mathematically what are we trying to perform.

1.3 The final form before the model creation

Until now we have assembled the code piece by piece while explaining what each cell is responsible for, but we haven't given a clear vision to the reader as to what we are trying to achieve - since it wasn't that necessary in order to understand the code. But now we start establishing more mathematically abstract structures hence it is clearly of utmost importance that the reader is able to follow the ideas behind the somewhat confusing implementation. Let us define some concepts.

In the concepts below we are going to be using the generalized terms 'user' and 'item'. We are clarifying here that **user refers to playlist** and **item refers to track**

2 basic techniques

In recommendation systems academic literature, no matter which implementation one decides to choose, there are 2 main techniques available for usage; Collaborative and content-based filtering. Firstly let's explain the latter and why it will prove hard and unproductive to implement. Content-based filtering methods are based on a description of the item and a profile of the user's preferences.[wikipedia] These methods are best suited to situations where there is known data on an item (name, location, description, etc.), but not on the user. What does that mean in our situation? It means we need meta data about each track, some features, individual characteristics. Since we are not provided with any of the above-mentioned we are abandoning the idea of using a content-based technique.

Collaborative filtering

Collaborative filtering is based on the assumption that people who agreed in the past will agree in the future, and that they will like similar kinds of items as they liked in the past. The system generates recommendations using only information about rating profiles for different users or items. By locating peer users/items with a rating history similar to the current user or item, they generate recommendations using this neighborhood.[wikipedia]. What does that mean? It means that if a playlist A has 50 tracks and playlist B has 30 tracks and they have say 25 tracks in common, there is a high chance we can take the tracks from playlist A (while respecting the order!!!) and present them as a prediction for playlist B. **Utility Matrices**

We have seen above that in a recommendation system application there are two classes of entities, users and items. Users have preferences for certain items, and these preferences must be teased out of the data. The data itself is represented as a utility matrix, giving for each user-item pair, a value that represents what is known about the degree of preference of that user for that item. In our case we will count as preference the occasion where a track belongs to a playlist. Think of a 2d array with each row to be a playlist and each column a track and take the (i,j)th element to be 1 when the jth track belongs to the ith playlist. Given this chance, we are going to depict the logic behind the whole structure we are going to use in order to predict tracks to playlists. Let's say we have been given the dataset

$$M = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{1}$$

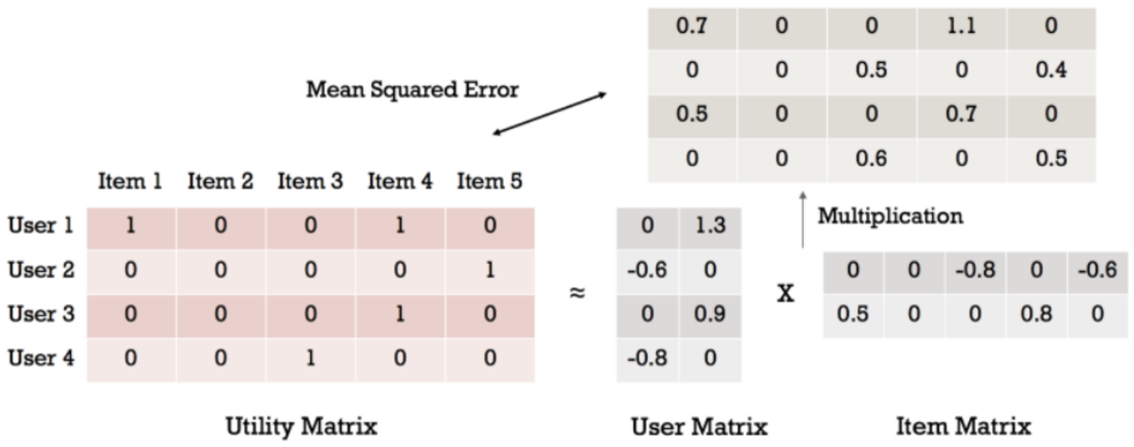
Above we have 3 playlists (P1, P2, P3) and 5 tracks (T1, T2, T3, T4, T5). Also we have

$$P1 \in (\ T1 \quad T4\ ), P2 \in (\ T2 \quad T4\ ), P3 \in (\ T1 \quad T2 \quad T5\ ) \tag{2}$$

and we also know, that in *some* playlists *some* tracks have been removed. We have been asked to fill the playlists with the tracks we actually believe that are included into them. As soon as the information is depicted into a utility matrix, we can start making suggestions using collaborative filtering methodology; We can say for example that P1 might also include T2 because both playlists have T4 included so *'they are similar'*. We can also do the same for P2 and T1. Afterwards we can fill both P1 and P2 with T5 and P3 with T4 on the grounds that *all three playlists started looking similar*. That simple and arguably naive example is designed to give you an idea about how CNF (Collaborative Neural Filtering) works.

Matrix factorization - its use on prediction

After composing the utility matrix, we can divide them into 2 separate ones(typically called user and item matrices). Now those new sub matrices are created with the idea that when you multiply them (remember, order MATTERS!) you get the original matrix. The initial utility matrix must be factorized in a way that the loss between it and the recreated one is minimized using **MSE**. Figure below explains the idea:



	Item 1	Item 2	Item 3	Item 4	Item 5
User 1	1	0	0	1	0
User 2	0	0	0	0	1
User 3	0	0	0	1	0
User 4	0	0	1	0	0

0	1.3
-0.6	0
0	0.9
-0.8	0

0	0	-0.8	0	-0.6
0.5	0	0	0.8	0

0.7	0	0	1.1	0
0	0	0.5	0	0.4
0.5	0	0	0.7	0
0	0	0.6	0	0.5

You may notice non zero values in previous cells that were tagged zero; the higher the value on an unobserved entry, the higher the chance the corresponding user will interact with the corresponding item! That is acquisition of new information and happens on the prediction process(where the 2 matrices try to recreate the original one). The idea behind this implementation is that each user and item is projected onto a latent space. We will explain more in detail later what a latent space is, but for now you can think of it as a way to represent compressed data. We can now measure the similarity between each latent vector by computing dot product or cosine similarity between them. In prediction we compute each of the user and item latent vector. Hence we will have:

$$\begin{pmatrix} 0 & 1.3 \end{pmatrix} X \begin{pmatrix} 0 \\ 0.5 \end{pmatrix}, \begin{pmatrix} 0 & 1.3 \end{pmatrix} X \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

$$\begin{pmatrix} 0 & 1.3 \end{pmatrix} X \begin{pmatrix} -0.8 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 & 1.3 \end{pmatrix} X \begin{pmatrix} 0 \\ 0.8 \end{pmatrix}, \begin{pmatrix} 0 & 1.3 \end{pmatrix} X \begin{pmatrix} -0.6 \\ 0 \end{pmatrix},$$

for the first row of the new utility matrix. And mathematically this is what happens during prediction:

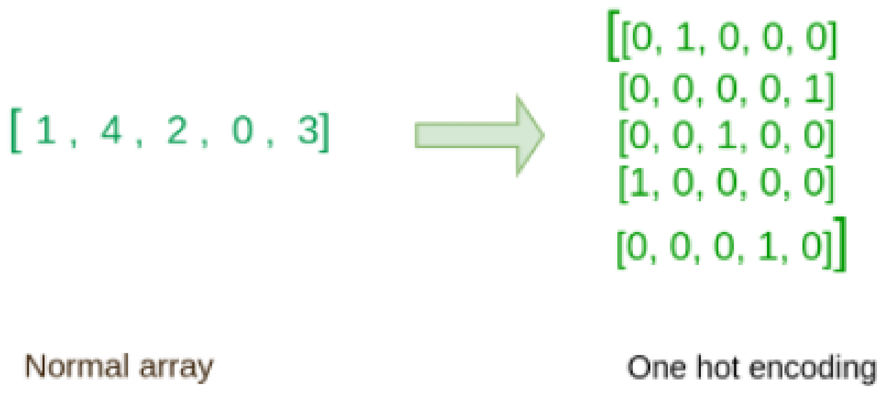
$$\hat{y}_{ui} = f(u, i | \mathbf{p}_u, \mathbf{q}_i) = \mathbf{p}_u^T \mathbf{q}_i = \sum_{k=1}^{K} p_{uk} q_{ik}$$

The prediction equals the inner product of latent vectors

The prediction of each user/item entry is merely the dot product of their latent vectors. Now we will discuss the reason why the above technique is not sufficient and how we can generalize it.

Generalizing matrix factorization

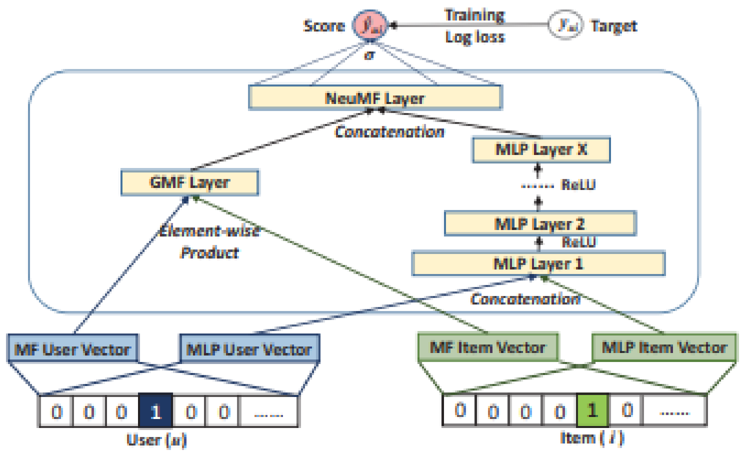
The inner product does not accurately predict the users/items relationships as it can be seen here. Thus we need a non linear approach to cover for the high complexity of the problem. One approach is to use a large number of latent factors, but it is easily dismissed because it can potentially damage the model generalization, leading to overfitting. The final solution is to create a deep neural network (dnn) to learn the user/items interactions. The key is to ensemble a model that utilises both the strength of MF linearity and the dnn non-linearity for modelling the user/item latent structures. In order for that to succeed, we first use one-hot-encoding of user/item of the input.



Because of how neural networks work, the data is depicted in high dimensions. We will explain in the NeuMF section more details about how lowering the dimensions work, but for now keep in mind that Matrix factorization is used on the final stages of the model, using as inputs vectors that are already processed by the dnn. **Please note** that GMF and MLP are not fused by sharing the same layer as input and their outputs are NOT combined together. If you are interested in this type of architecture you can check Neural Tensor Networks(NTNs). The architecture above cannot be implemented here because:

1. It implies that both GMF and DNN use the same size of the lowered dimensions.
2. It lowers flexibility

Hence we allow both GMF and MLP (the name of dnn) to learn separately how to lower the dimensions (i.e have their own embeddings) and then combine the 2 models by concatenating their last layer. This is a full image of the architecture:

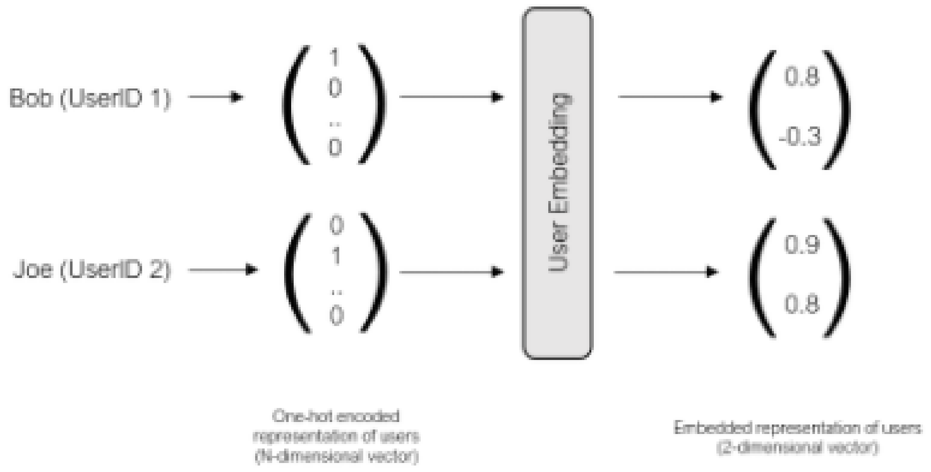


Observe that both models work simultaneously and independently from each other.

2 Creating the model

2.1 NeuMF

The model is a neural network. In this paper we are assuming the reader is familiar to how a neural network works, for, we will not explain terminology such as 'epochs' or 'back-propagation'. We are though, going to describe layer by layer what the network is performing -to the extend possible (do not forget deep learning is infamous for it's low interpretability). NeuMF is designed to be a connection of GMF and MLP. In the input layer we have the user's and item's multidimensional tensors. We are passing them into the embedding layer which performs a dimensionality reduction, in practice, an abstraction whose purpose is for the model to find some patterns that will help the learning process.An embedding is a relatively low-dimensional space into which you can translate high-dimensional vectors.


Bob (UserID 1)
Joe (UserID 2)
User Embedding
0.8
-0.3
0.9
0.8
One-hot encoded representation of users (N-dimensional vector)
Embedded representation of users (2-dimensional vector)

Embeddings make it easier to do machine learning on large inputs like sparse vectors representing words. Ideally, an embedding captures some of the semantics of the input by placing semantically similar inputs close together in the embedding space. An embedding can be learned and reused across models. [**Embeddings by google**] We create both MF and MLP embedding layers.

Next we are Flattening the embeddings into the MF and MLP latent vectors. Flattening is converting the data into a 1-dimensional array for inputting it to the next layer. Latent vectors are a bunch of latent-inaccessible variables. That latent data is 'hidden' from us, that is, we don't know in which way the network will map the data to lower dimensions and what these lower dimensions will mean. We trust the network has trained itself for performing the best configuration.

It's time we add the 5 Dense layers. Until now we were constructing -and perfecting- the correct input form. Now it is time to set those parameters (weights) that are going to understand that, say, playlist no 3400 has the same 50 out of it's 60 tracks with playlists no 5600 50 out of 55 tracks. Note that we only use the MLP latent vector which is a concatenation of MLP user and item vectors respectfully. The activation function is **ReLU**.

We talked earlier that this model is a connection of performing Generalized Matrix factorization and the Multilayer Perceptron. The connection arrives in this layer, the predict layer where a concatenation of mf latent and mlp vector (that has been through the exhausting Dense layers).

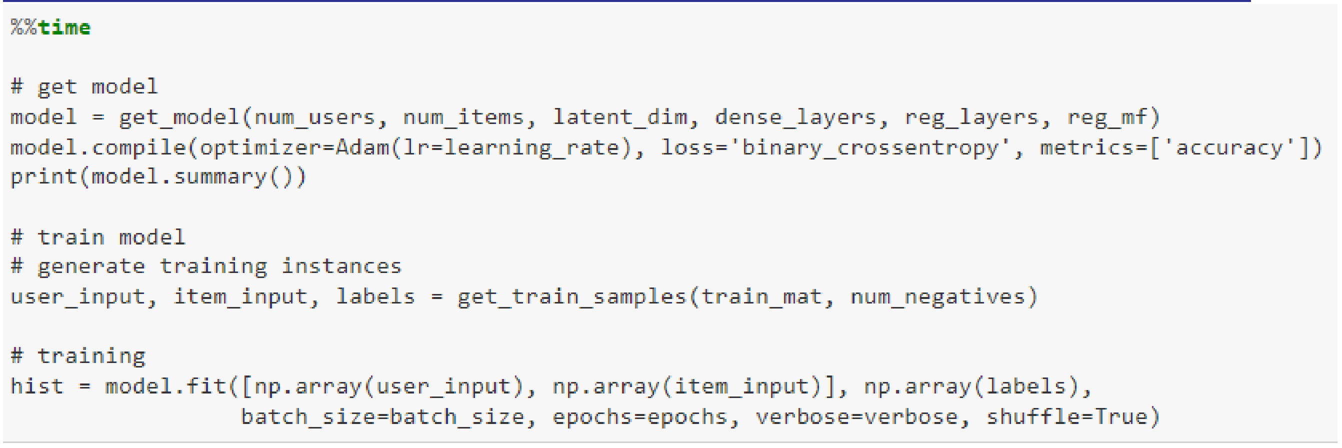
As it is common in classification problems models, we add the final layer to be a single neuron with the sigmoid as an activation function. That decision is supported by the use of binary cross-entropy(i.e the difference 2 probability distributions) as a loss function.

Below we present the code we just explained:

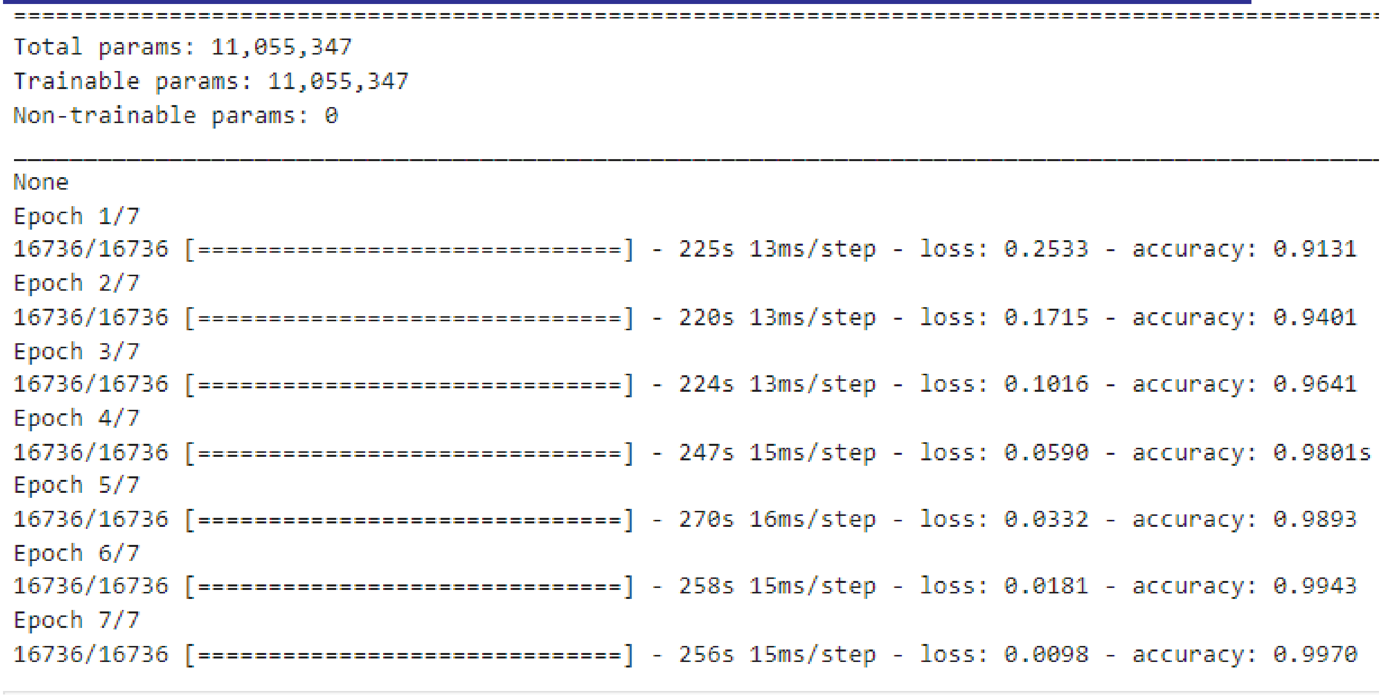
```python
# full NCF model
def get_model(num_users, num_items, latent_dim=8, dense_layers=[64, 32, 16, 8],
              reg_layers=[0, 0, 0, 0], reg_mf=0):

    # input layer
    input_user = Input(shape=(1,), dtype='int32', name='user_input')
    input_item = Input(shape=(1,), dtype='int32', name='item_input')

    # embedding layer
    mf_user_embedding = Embedding(input_dim=num_users, output_dim=latent_dim,
                        name='mf_user_embedding',
                        embeddings_initializer='RandomNormal',
                        embeddings_regularizer=l2(reg_mf), input_length=1)
    mf_item_embedding = Embedding(input_dim=num_items, output_dim=latent_dim,
                        name='mf_item_embedding',
                        embeddings_initializer='RandomNormal',
                        embeddings_regularizer=l2(reg_mf), input_length=1)
    mlp_user_embedding = Embedding(input_dim=num_users, output_dim=int(dense_layers[0]/2),
                         name='mlp_user_embedding',
                         embeddings_initializer='RandomNormal',
                         embeddings_regularizer=l2(reg_layers[0]),
                         input_length=1)
    mlp_item_embedding = Embedding(input_dim=num_items, output_dim=int(dense_layers[0]/2),
                         name='mlp_item_embedding',
                         embeddings_initializer='RandomNormal',
                         embeddings_regularizer=l2(reg_layers[0]),
                         input_length=1)

    # MF latent vector
    mf_user_latent = Flatten()(mf_user_embedding(input_user))
    mf_item_latent = Flatten()(mf_item_embedding(input_item))
    mf_cat_latent = Multiply()([mf_user_latent, mf_item_latent])

    # MLP latent vector
    mlp_user_latent = Flatten()(mlp_user_embedding(input_user))
    mlp_item_latent = Flatten()(mlp_item_embedding(input_item))
    mlp_cat_latent = Concatenate()([mlp_user_latent, mlp_item_latent])

    mlp_vector = mlp_cat_latent

    # build dense layer for model
    for i in range(1,len(dense_layers)):
        layer = Dense(dense_layers[i],
                      activity_regularizer=l2(reg_layers[i]),
                      activation='relu',
                      name='layer%d' % i)
        mlp_vector = layer(mlp_vector)

    predict_layer = Concatenate()([mf_cat_latent, mlp_vector])
    result = Dense(1, activation='sigmoid',
                   kernel_initializer='lecun_uniform',name='result')

    model = Model(inputs=[input_user,input_item], outputs=result(predict_layer))

    return model
```

And below is demonstrated the code needed to train the model, unluckily we could not use our GPU's to accelerate the whole process at that time. **get_train_samples** is a generative function, used to produce negative training examples(invalid user-item pairs). **HYPERPARAMETERS USED:**

- **loaded** True - the data are already loaded and ready for insertion
- **verbose** 1 - while validating we want the bar visualizing us the procedure with respect to time
- **epochs** 7 - one cycle through the full training dataset, bibliography suggests ideal is 11
- **batch_size** 256 -we are filling the network with chunks of data since the whole sparse matrix is of enormous size
- **latent_dim** 32 - Latent dimensions/latent variables are variables which we do not directly observe, but we assume to exist and save, or instruct the merged features from matrix factorization
- **dense_layers** [256, 128, 128, 64]- previously we mentioned we create 5 dense layers, each has their own number of neurons.
- **learning_rate** 0.001 - a classic one



```
%%time

# get model
model = get_model(num_users, num_items, latent_dim, dense_layers, reg_layers, reg_mf)
model.compile(optimizer=Adam(lr=learning_rate), loss='binary_crossentropy', metrics=['accuracy'])
print(model.summary())

# train model
# generate training instances
user_input, item_input, labels = get_train_samples(train_mat, num_negatives)

# training
hist = model.fit([np.array(user_input), np.array(item_input)], np.array(labels),
                 batch_size=batch_size, epochs=epochs, verbose=verbose, shuffle=True)
```



```
=================================================================================================
Total params: 11,055,347
Trainable params: 11,055,347
Non-trainable params: 0
_________________________________________________________________________________________________
None
Epoch 1/7
16736/16736 [==============================] - 225s 13ms/step - loss: 0.2533 - accuracy: 0.9131
Epoch 2/7
16736/16736 [==============================] - 220s 13ms/step - loss: 0.1715 - accuracy: 0.9401
Epoch 3/7
16736/16736 [==============================] - 224s 13ms/step - loss: 0.1016 - accuracy: 0.9641
Epoch 4/7
16736/16736 [==============================] - 247s 15ms/step - loss: 0.0590 - accuracy: 0.9801s
Epoch 5/7
16736/16736 [==============================] - 270s 16ms/step - loss: 0.0332 - accuracy: 0.9893
Epoch 6/7
16736/16736 [==============================] - 258s 15ms/step - loss: 0.0181 - accuracy: 0.9943
Epoch 7/7
16736/16736 [==============================] - 256s 15ms/step - loss: 0.0098 - accuracy: 0.9970
```

2.2 Clustering

The model is ready! Does that mean we initiate the predictions on 9 thousand playlists? The answer is not yet. Now the importance and utility of embedding vectors can shine. Remember the function of the embedding layer? We are exploiting that in order to perform clustering. Clustering in machine learning is the process of finding items with similar features (that are explored in embeddings layers) and grouping them together. In our case after clustering the playlists we will call two playlists that belong in the same cluster as 'neighbours'.

```
%%time
from sklearn.cluster import KMeans

mlp_user_embedding_weights = (next(iter(filter(lambda x: x.name == 'mlp_user_embedding',
                                               model.layers))).get_weights())

# Get laten embedding for desired user
user_latent_matrix = mlp_user_embedding_weights[0]

print('\nPerforming kmeans to find the nearest users/playlists')
kmeans = KMeans(n_clusters=250, random_state=0, verbose=0).fit(user_latent_matrix)
```

```
Performing kmeans to find the nearest users/playlists
Wall time: 35.3 s
```

3 Applying the Model

This is the final part. Here we use the trained model in order to generate results for the missing playlists. Before we look at the code though, let's review some rules of the competition. 1.For each playlist we must have exactly 500 tracks generated. 2.There should be no duplicates(all of the tracks must be unique). There are 2 instances where the 1st condition may not be correct; if the model predicts more or fewer than 500 tracks. The first case is can be easily handled by sorting(which we are going to apply anyway) and keeping the first 500 tracks. But in order to tackle the case where the model predicts say 400 tracks, can we perform something better than randomly adding 100 tracks that don't belong to the playlist? The answer is yes, we are going to fill the playlist with the most popular tracks because statistically there are higher chances those will be contained on the playlist. Following that idea we are writing the code below:

```
most_popular_songs = song_playlist_df['trackid'].value_counts().index.tolist()[:500]
```

Now, the whole process of creating the final list, ready for export into a csv file -which may be overwhelming- will be presented below, and following it, we are going to explain each part of the code individually.

```
%%time
#### Get csv for submition
final_list_to_csv = []
iter_counter = 0
times_milestone_reached = 0
for playlist_id in range(15000,19000):
    iter_counter+= 1
    if(iter_counter % 200 == 0):
        times_milestone_reached +=1
        print(str(times_milestone_reached*5)+"% completed")

    desired_user_id = playlist_id

    # Get laten embedding for desired user
    one_user_vector = user_latent_matrix[desired_user_id,:]
    one_user_vector = np.reshape(one_user_vector, (1,64))

    desired_user_label = kmeans.predict(one_user_vector)
    user_label = kmeans.labels_
    neighbors = []
    for user_id, user_label in enumerate(user_label):
        if user_label == desired_user_label:
            neighbors.append(user_id)
    #print('Found {0} neighbor users/playlists.'.format(len(neighbors)))

    tracks = []
    for user_id in neighbors:
        tracks += list(song_playlist_df[song_playlist_df['pid']
                                        == int(user_id)]['track_index'])
    #print('Found {0} neighbor tracks from these users.'.format(len(tracks)))

    users = np.full(len(tracks), desired_user_id, dtype='int32')
    items = np.array(tracks, dtype='int32')
```

```
#print('\nRanking most likely tracks using the NeuMF model...')
# and predict tracks for my user
results = model.predict([users,items],batch_size=100, verbose=0)
results = results.tolist()
#print('Ranked the tracks!')

results_df = pd.DataFrame(np.nan, index=range(len(results)),
                          columns=['probability','track_uri'])
#print(results_df.shape)

# Connect tracks names with the probability that user will like them
for i, prob in enumerate(results):
    results_df.loc[i] = [prob[0], song_playlist_df[song_playlist_df['track_index']
                                                   == tracks[i]].iloc[0]['trackid']]
results_df = results_df.sort_values(by=['probability'], ascending=False)

#deleting duplicates and tracks that alaready exist in playlist
temp = song_playlist_df[song_playlist_df['pid'] == desired_user_id]
temp = temp['trackid']

results_df = results_df.drop_duplicates(subset=['track_uri'])
#print(len(results_df))


results_df = results_df[~results_df.track_uri.isin(temp)]
#len(results_df)

songs_that_exist_in_playlist = temp.values.tolist()

#results_df = results_df[:500]
results_df = results_df[0:500]['track_uri'].values.tolist()

if (len(results_df) < 500):
    temp_most_popular = list(set(most_popular_songs) -
                             (set(results_df) + set(songs_that_exist_in_playlist)))
    results_df.extend(temp_most_popular)
    results_df = results_df[:500]
    temp_list = [str(challenge_set_pid_last_9000[playlist_id - 9000])]
    temp_list.extend(results_df)
    #for index,row in results_df.iterrows():
     #   temp_list.append(row['track_uri'])
    final_list_to_csv.append(temp_list)
```

```
5% completed
10% completed
15% completed
20% completed
25% completed
30% completed
35% completed
40% completed
45% completed
50% completed
55% completed
60% completed
65% completed
70% completed
75% completed
80% completed
85% completed
90% completed
95% completed
100% completed
Wall time: 9h 32min 16s
```

```
print(len(final_list_to_csv))
```

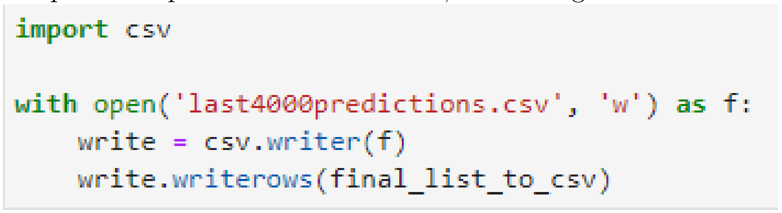
```
4000
```

Let's begin the explanation. Variables iter_counter and times_milestone_reached are used for feedback about the percentage of predictions that is completed. Hence we see 30% completed for example.

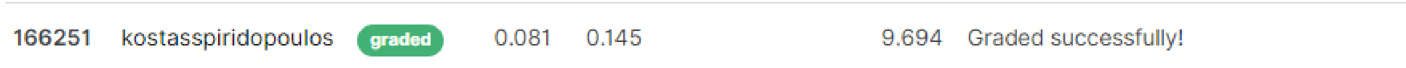
The for loop. In this instance we are going to predict just a chunk of playlists. The variable desired_user_id is the current playlist we are working on. We are getting the specific user's vector from the whole latent matrix and we are reshaping it in order to fit it correctly into the model. Then we are performing kmeans in order to find the neighbours of that user(users that have same latent vectors which means same characteristics). *Reminder:* Users reference playlists - items tracks. Next, after having found the neighbours, we are delving into the tracks. For the neighbours of the desired user we are extracting the neighbour tracks. But how is a neighbour track defined? This is occurring inside the for loop. Afterwards we are creating *users* and *items*. User is a numpy array filled using the full method. What's happening is that users is a numpy array of size of the length of tracks(which are the neighbouring tracks of the particular playlist) and every position is filled in with the playlist id we are currently filling. items is another numpy array containing neighbouring tracks. We are feeding them into the model for prediction. This ranks the "neighboring" tracks by how likely they are (or how the model believes) to occur next in the given test playlist. We are converting them into a pandas dataframe and get the probability of being in the playlist ()according to the model), the track index and the track id through looping into the results. Following the 2nd rule, we are removing the duplicates or tracks that already existed in the playlist. temp holds the playlists that were already in the playlist. Then following the 1st rule, we are only keeping the first 500 tracks. But remember there is another instance where the 1st rule may be violated. What may happen if there are less than 500 tracks? As described above, we are filling the remaining tracks with the most popular ones(while keeping an eye for duplicates again with the variable *songs_that_exist_in_playlist*). Last but not least, we are putting the information in a list, along with the playlist in demand.

4 Preparation for submission

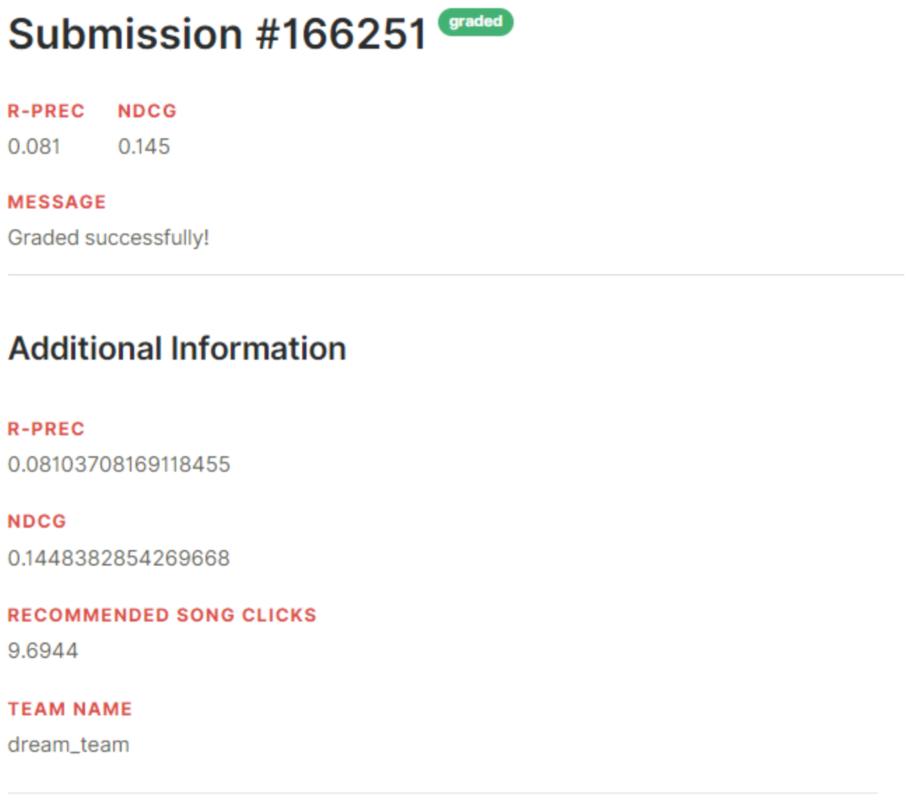
This is it! We have the predictions ready. It was an enjoyable (and a long -took nine and a half hours just for this chunk of playlists-) journey. Now all that's left is to write them into a csv file following the competition's protocol. And of course, submitting them!



```
import csv

with open('last4000predictions.csv', 'w') as f:
    write = csv.writer(f)
    write.writerows(final_list_to_csv)
```



```
166251    kostasspiridopoulos    graded    0.081    0.145        9.694    Graded successfully!
```



Submission #166251 graded

R-PREC 0.081
NDCG 0.145

MESSAGE
Graded successfully!

Additional Information

R-PREC
0.08103708169118455

NDCG
0.1448382854269668

RECOMMENDED SONG CLICKS
9.6944

TEAM NAME
dream_team