In [ ]:
```
#!pip install spotipy
```

In [2]:
```
import re
import itertools
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
from spotipy.oauth2 import SpotifyOAuth
import spotipy.util as util

from skimage import io
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from datetime import datetime
import plotly
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots


from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import RandomizedSearchCV, cross_val_score, tr
from sklearn.linear_model import LinearRegression, ElasticNetCV
from sklearn.ensemble import RandomForestRegressor, VotingRegressor, Stacki
from xgboost import XGBRegressor

import warnings
warnings.filterwarnings("ignore")
```

The datasets used can be found on Kaggle at: https://www.kaggle.com/vatsalmavani/music-recommendation-system-using-spotify-dataset/data. In particular, "data.csv" has been used to perform the analysis relative to the Popularity prediction of songs. For what concerns the Recommender System, also "data_w_genres.csv" has been used to extract and combine the information regarding the genres.

Spotify Songs Popularity Prediction

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
import os
os.chdir('/content/drive/MyDrive/Colab Notebooks')
```

In [ ]:
```
df = pd.read_csv('data.csv')
df.head()
```

Out[ ]:

	valence	year	acousticness	artists	danceability	duration_ms	energy	explicit
0	0.0594	1921	0.982	['Sergei Rachmaninoff', 'James Levine', 'Berli...	0.279	831667	0.211	0
1	0.9630	1921	0.732	['Dennis Day']	0.819	180533	0.341	0
2	0.0394	1921	0.961	['KHP Kridhamardawa Karaton Ngayogyakarta Hadi...	0.328	500062	0.166	0
3	0.1650	1921	0.967	['Frank Parker']	0.275	210000	0.309	0
4	0.2530	1921	0.957	['Phil Regan']	0.418	166693	0.193	0

In [ ]:
```
df.describe()
```

Out[ ]:

	valence	year	acousticness	danceability	duration_ms	
count	170653.000000	170653.000000	170653.000000	170653.000000	1.706530e+05	17065
mean	0.528587	1976.787241	0.502115	0.537396	2.309483e+05	
std	0.263171	25.917853	0.376032	0.176138	1.261184e+05	
min	0.000000	1921.000000	0.000000	0.000000	5.108000e+03	
25%	0.317000	1956.000000	0.102000	0.415000	1.698270e+05	
50%	0.540000	1977.000000	0.516000	0.548000	2.074670e+05	
75%	0.747000	1999.000000	0.893000	0.668000	2.624000e+05	
max	1.000000	2020.000000	0.996000	0.988000	5.403500e+06	

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170653 entries, 0 to 170652
Data columns (total 19 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   valence           170653 non-null  float64
 1   year              170653 non-null  int64  
 2   acousticness      170653 non-null  float64
 3   artists           170653 non-null  object 
 4   danceability      170653 non-null  float64
 5   duration_ms       170653 non-null  int64  
 6   energy            170653 non-null  float64
 7   explicit          170653 non-null  int64  
 8   id                170653 non-null  object 
 9   instrumentalness  170653 non-null  float64
 10  key               170653 non-null  int64  
 11  liveness          170653 non-null  float64
 12  loudness          170653 non-null  float64
 13  mode              170653 non-null  int64  
 14  name              170653 non-null  object 
 15  popularity        170653 non-null  int64  
 16  release_date      170653 non-null  object 
 17  speechiness       170653 non-null  float64
 18  tempo             170653 non-null  float64
dtypes: float64(9), int64(6), object(4)
memory usage: 24.7+ MB
```

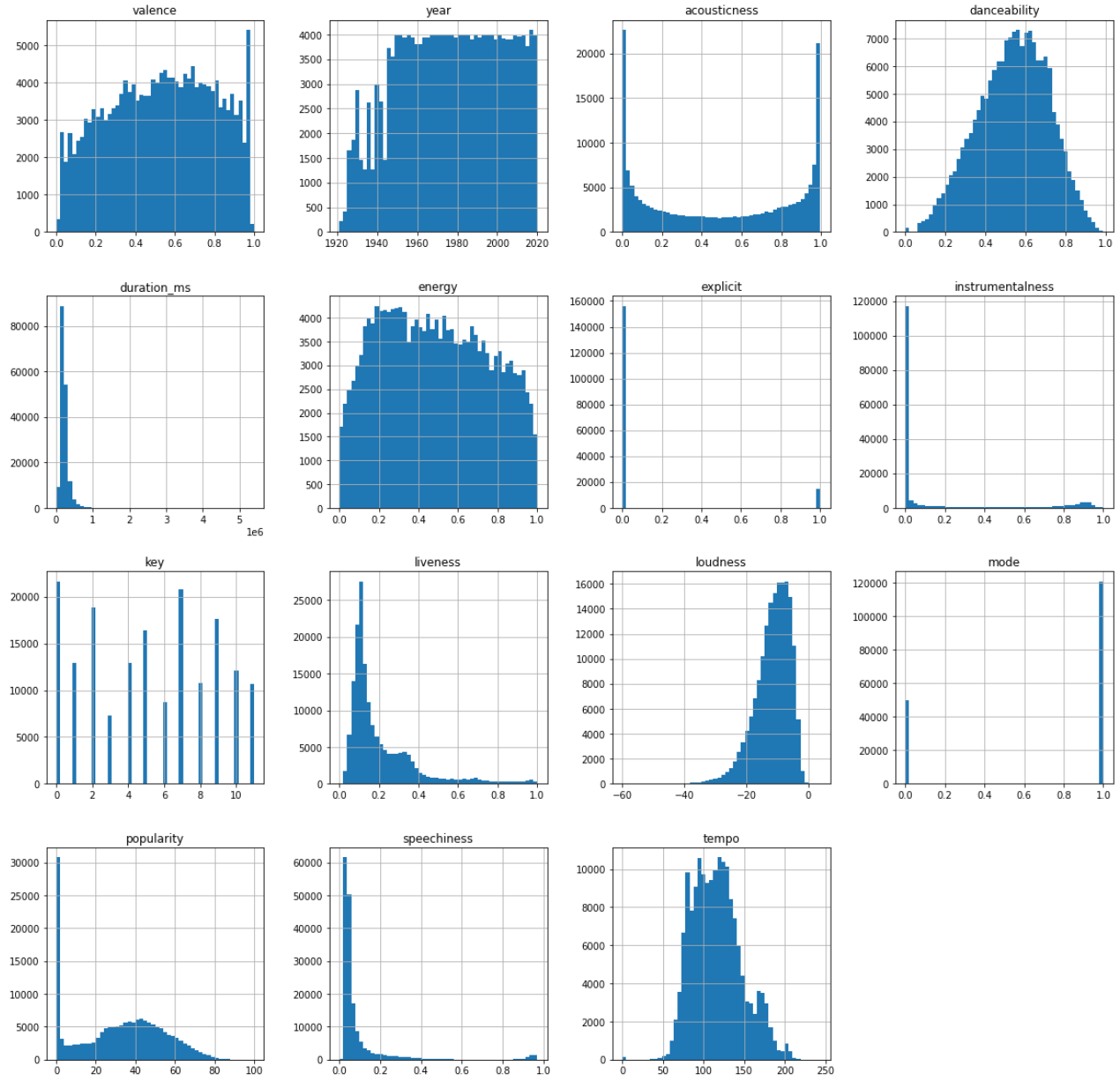
Exploratory Data Analysis

Here I am plotting the histograms of the features in the dataset, to have a rough idea of their distributions. In particular, we can notice that:

- We have less data regarding songs released before approx. 1950, which might imply that our models might be less accurate in the timeframe 1920 - 1950.
- Danceability presents a distribution that resembles the normal one, centered at about 0.55.
- Duration, Instrumentalness and Speechiness are left-skewed with a peak at about 0.1.
- Loudness is mostly contained in the range [-20,0], but a deeper analysis of this phenomena is given later in the notebook.
- Explicit and Mode are binary features.
- Popularity is our target variable and will be deeply explored in the following cells.
- Acousticness presents two peaks at about 0 and 1, but I cannot infer any analysis at this point.
- Key is a categorical variable and its relation with popularity will be explored in the following cells.

In [ ]:

```
df.hist(bins = 50, figsize = (20,20))
plt.show()
```



In the following cell, I am plotting the boxplots of some chosen features to further explore the presence of outliers. In addition, the plots are generated with plotly library which allow us to explore interactvely the values of the bloxplots (other than looking nice).

In [4]:

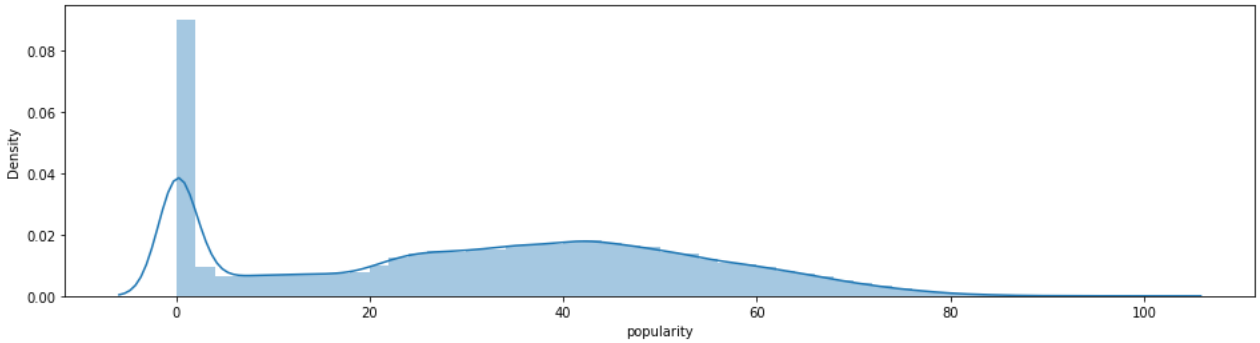
```
box_cols = ['danceability', 'energy', 'key', 'loudness', 'speechiness', 'ac
rows, cols = 3, 4
fig = make_subplots(rows=rows, cols=cols, subplot_titles=box_cols)
x, y = np.meshgrid(np.arange(rows)+1, np.arange(cols)+1)
count = 0
for row, col in zip(x.T.reshape(-1), y.T.reshape(-1)):
    try:
        fig.add_trace(
            go.Box(x = df[box_cols[count]].values, name=''),
            row = row,
            col = col
        )
        count+=1
    except:
        break

fig.update_layout(height=900, width=900, title_text='Boxplot Distribution',
fig.show()
```

From the following plot, we can notice that the distribution of popularity is left-skewed with a peak at about popularity equal to zero.

In [ ]:
```
plt.figure(figsize=(16, 4))
sns.distplot(df["popularity"])
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f58e7080c90>
```



The following correlation heatmap allows us to explore the correlation between our target variable (popularity) and the other features of the dataset. In particular, we can notice that the most positively correlated features with Popularity are:

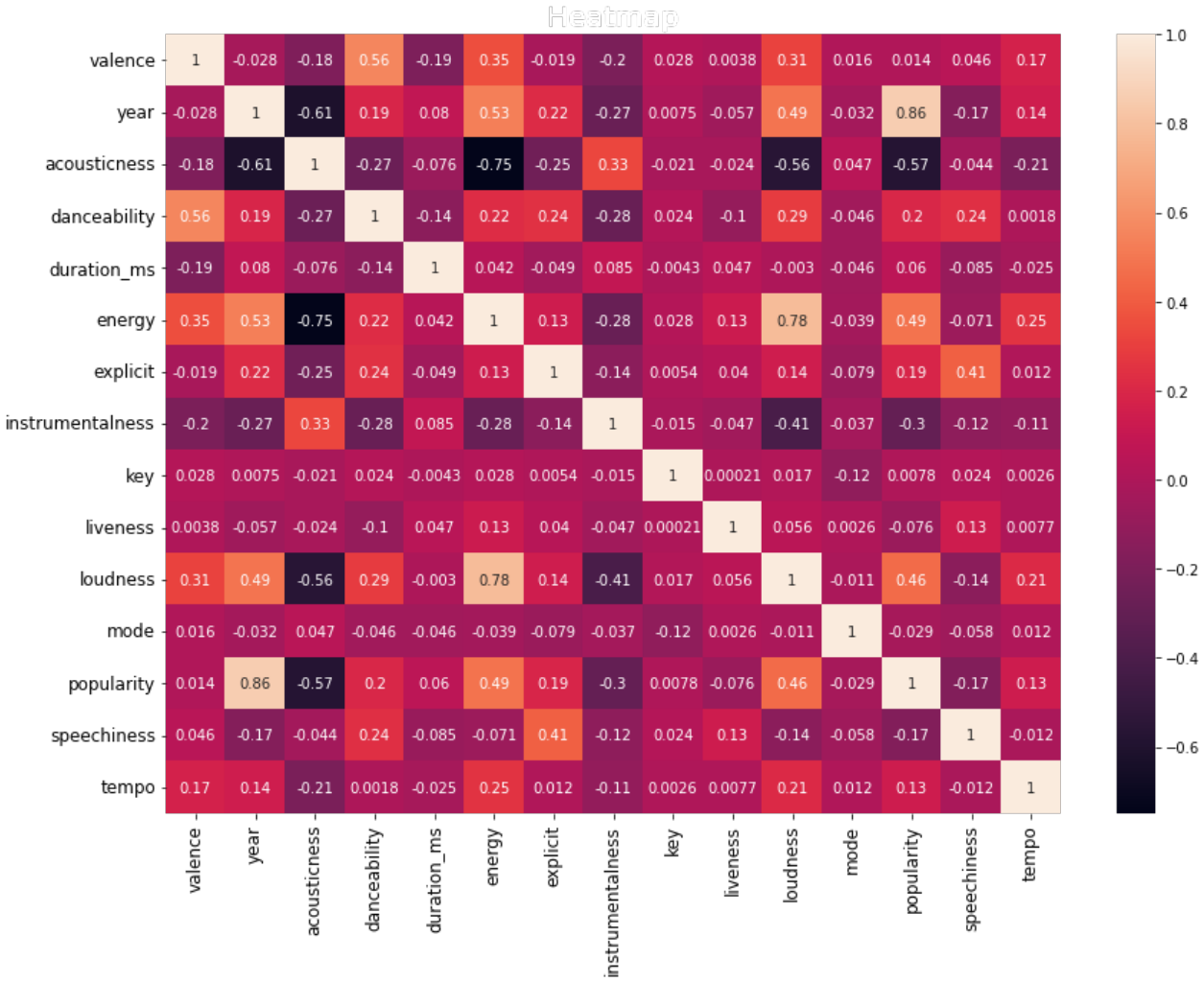
- Year
- Energy
- Loudness

While among the negative correlations we have:

- Acousticness
- Instrumentalness
- Speechiness

Finally, the least correlated feature is: Key.

In [ ]:
```
plt.figure(figsize = (14, 10))
sns.heatmap(df.corr(), annot=True)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.title('Heatmap', fontsize=20, color='white')
plt.show()
```

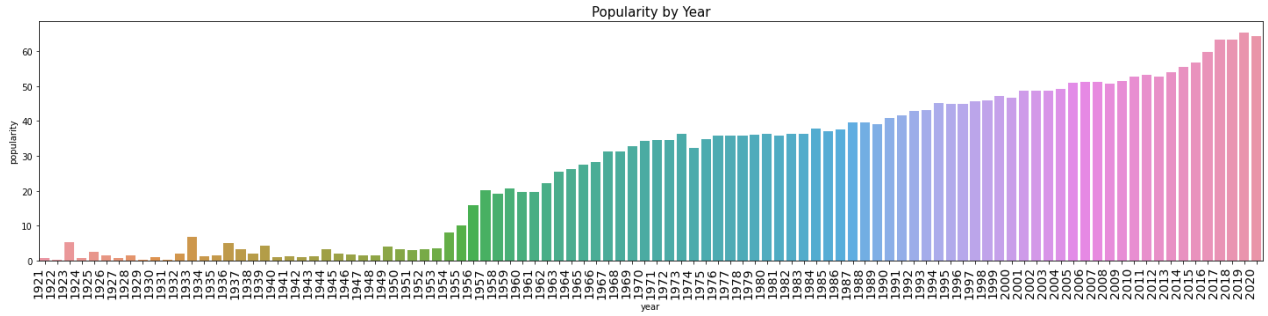


Here I am inspecting the relation between Popularity and Year of release of the songs. As we will deeper discuss later, we see an increasing trend in the popularity of the songs over the years, this is probability due to a simultaneous increase in popularity of Spotify.

In [ ]:

```
count = df.groupby('year')[['popularity']].count().sort_values(by='popularity'
sum = df.groupby('year')[['popularity']].sum().sort_values(by='popularity',
rank = sum.popularity / count.popularity
rank = rank.sort_values(ascending=True)
rank = pd.DataFrame(rank).reset_index()

plt.figure(figsize = (25, 5))
sns.barplot(x='year', y='popularity', data=rank)
plt.xticks(fontsize=14, rotation=90, ha="right")
plt.title('Popularity by Year',fontsize=15, color='black')
plt.show()
```

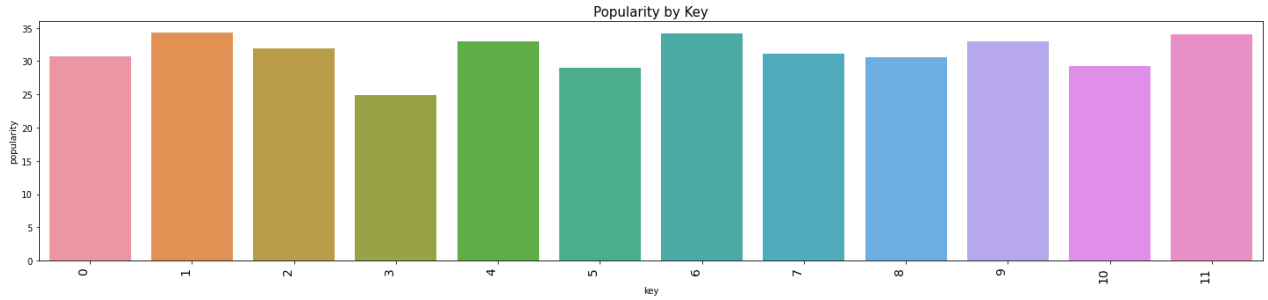


As we can see from the plot below, key is effectively the least correlated feature with popularity. Indeed, no real trend can be established.

In [ ]:

```
count = df.groupby('key')[['popularity']].count().sort_values(by='popularit
sum = df.groupby('key')[['popularity']].sum().sort_values(by='popularity',
rank = sum.popularity / count.popularity
rank = rank.sort_values(ascending=True)
rank = pd.DataFrame(rank).reset_index()

plt.figure(figsize = (25, 5))
sns.barplot(x='key', y='popularity', data=rank)
plt.xticks(fontsize=14, rotation=90, ha="right")
plt.title('Popularity by Key',fontsize=15, color='black')
plt.show()
```



Data Cleaning

We can check for the presence of NaN values, since we have no missing data we can avoid to perform any imputation

In [ ]:

```
pd.isnull(df).sum()
```

Out[ ]:
```
valence             0
year                0
acousticness        0
artists             0
danceability        0
duration_ms         0
energy              0
explicit            0
id                  0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
name                0
popularity          0
release_date        0
speechiness         0
tempo               0
dtype: int64
```

From a quick look at the boxplots, we can notice that the following features might be affected by outliers:

- "Loudness", since humans are not capable of distinguishing frequencies approximately below 20Db, I will clip any value below -20 to -20.
- "Liveness", it defines a probability estimation of whether the track was recorded in studio or live, hence I will encoded it as <= 0.5 (Studio), > 0.5 (Live).
- "Duration_ms" and "Speechness", since audiobooks and movies are also present in this dataset, I will drop the outliers.

Hence, we proceed in the cleaning of these features. We start with a dataset of about 170k songs and drop to approximately 155k after dropping entries according to previosuly mentioned conditions.

In [ ]:
```
np.shape(df)
```

Out[ ]:
```
(170653, 19)
```

In [ ]:
```
def cleaning(df):
    df['loudness'] = df['loudness'].clip(-20,0)
    df = df[df['duration_ms'] <= 650000]
    df = df[df['speechiness'] <= 0.25]
    df = df.drop(columns=['id', 'artists', 'name', 'release_date'])
    return df
```

In [ ]:
```
cat = np.zeros(len(df['liveness']))
for i in range(len(df['liveness'])):
    if df['liveness'][i] > 0.5:
        cat[i] = 1
    else:
        cat[i] = 0
df['liveness'] = cat
```

In [ ]:
```
df = cleaning(df)
np.shape(df)
```

Out[ ]:
```
(155465, 15)
```

In [ ]:
```
df.head()
```

Out[ ]:

	valence	year	acousticness	danceability	duration_ms	energy	explicit	instrumentalnes
2	0.0394	1921	0.961	0.328	500062	0.166	0	0.91300
3	0.1650	1921	0.967	0.275	210000	0.309	0	0.00002
4	0.2530	1921	0.957	0.418	166693	0.193	0	0.00000
5	0.1960	1921	0.579	0.697	395076	0.346	0	0.16800
6	0.4060	1921	0.996	0.518	159507	0.203	0	0.00000

Then I am going to drop the columns that are either not useful or not exploitable for the prediction taks, i.e., "artists", "id", "name".

Then I move to the evaluation of three different models. In particular, we are dealing with a regression task, we are regressing the features of the dataset to predict the popularity of a song. In order to do so, I will use LinearRegression as a baseline model, as it represents the simplest model and building block of regression analysis. Then I will implement XGBRegressor, i.e. a very robust boosting algorithm which is often exploited in Data Science competitions due to its ability of achieving good performance with relatively simple hyperparameters optimization. In addition, I also try a RandomForestRegressor, i.e., a tree-based decision algorithm that exploits bagging and randomness. I evaluate the performance of this three algorithms according to Mean Squared Error.

In [ ]:
```
features = df.drop(columns='popularity')
targets = df['popularity']
X_train, X_test, y_train, y_test = train_test_split(features, targets, test
```

In [ ]:
```
regressors = [LinearRegression(),
              XGBRegressor(),
              RandomForestRegressor()]

for model in regressors:
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    error = mean_squared_error(y_test, predictions)
    print(f'MSE {type(model).__name__}: {error}')
```

```
MSE LinearRegression: 119.17056337304089
[09:38:14] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:lin
ear is now deprecated in favor of reg:squarederror.
MSE XGBRegressor: 99.14883747707412
MSE RandomForestRegressor: 95.67950376337458
```

In the following cells, I will exploit RandomizedSearchCV to do the hyperparameters tuning of RandomForest and XGBoost. Then I will extract the results of the search and run the tuned models to asses eventual performance improvements.

The tuning of Random Forest took about 2 hours.

In [ ]:
```
#RANDOM FOREST TUNING:
param_grid = {'max_depth': np.arange(4, 21),
              'min_samples_split': [0.001, 0.01, 0.1, 2],
              'min_samples_leaf': [0.001, 0.01, 0.1, 1],
              'max_features': np.arange(3, 7),
              'n_jobs': [-1]
              }

rf = RandomForestRegressor()
random_search = RandomizedSearchCV(rf, param_distributions=param_grid, n_it
random_search.fit(features, targets)
```

In [ ]:
```
random_search.best_params_
```

Out[ ]:
```
{'max_depth': 16,
 'max_features': 3,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'n_estimators': 630,
 'n_jobs': -1}
```

The tuning of XGBoost took about 2 hours.

In [ ]:
```
# XGB TUNING:
param_grid = {'learning_rate': np.arange(0.01, 0.1, 0.01),
              'n_estimators': np.arange(2, 500, 10),
              'subsample': [0.7, 0.8, 0.9, 1.0],
              'max_depth': np.arange(4,21),
              'min_samples_split': [0.001, 0.01, 0.1, 2],
              'min_samples_leaf': [0.001, 0.01, 0.1, 1],
              'n_jobs': [-1]
              }

xgb = XGBRegressor()
random_search = RandomizedSearchCV(xgb, param_distributions=param_grid, n_i
search = random_search.fit(features, targets)
```

In [ ]:
```
search.best_params_
```

Out[ ]:
```
{'learning_rate': 0.01,
 'max_depth': 15,
 'min_samples_leaf': 1,
 'min_samples_split': 0.01,
 'n_estimators': 352,
 'n_jobs': -1,
 'subsample': 0.7}
```

In [ ]:
```
rf = RandomForestRegressor(n_estimators = 550,
                           max_depth = 16,
                           max_features = 3,
                           min_samples_leaf = 1,
                           min_samples_split = 2,
                           random_state = 42,
                           n_jobs = -1)

xgb = XGBRegressor(learning_rate = 0.01,
                   n_estimators = 352,
                   max_depth = 15,
                   min_samples_leaf = 1,
                   min_samples_split = 0.001,
                   subsample = 0.7)
```

In [ ]:
```
rf.fit(X_train, y_train)
predictions = rf.predict(X_test)
error = mean_squared_error(y_test, predictions)
print(error)
```

```
95.28303083666593
```

In [ ]:
```
rf.feature_importances_
```

Out[ ]:
```
array([0.01831053, 0.55501229, 0.14667729, 0.02077873, 0.03266366,
       0.06429057, 0.0146338 , 0.03972184, 0.00607761, 0.00136066,
       0.06558267, 0.00154397, 0.01957527, 0.0137711 ])
```

In [ ]:
```
xgb.fit(X_train, y_train)
predictions = xgb.predict(X_test)
error = mean_squared_error(y_test, predictions)
print(error)
```

```
[00:48:11] WARNING: /workspace/src/objective/regression_obj.cu:152: reg:lin
ear is now deprecated in favor of reg:squarederror.
93.98272653167982
```

In [ ]:
```
xgb.feature_importances_
```

Out[ ]:
```
array([0.0064876 , 0.81463695, 0.01169451, 0.01291995, 0.01670863,
       0.01369372, 0.02297889, 0.01739147, 0.01114008, 0.01591729,
       0.01598219, 0.01134251, 0.0142895 , 0.01481673], dtype=float32)
```

In the following cells, I eveluate through cross-validation (performed by KFold) the performance of our models, to assess the robustness of the architectures and avoid overfitting problems.

In [ ]:
```
estimators = [('rf', rf), ('xgb', xgb)]
kf = KFold(shuffle=True, random_state=42)
```

In [ ]:
```
for _, estimator in estimators:
    scores = cross_val_score(estimator, features, targets, scoring='neg_mea
    print(f'{type(estimator).__name__}: scores: {scores}, avg: {np.mean(sc
```

```
RandomForestRegressor: scores: [-95.36311828 -97.29784141 -94.27367605 -93.
82199808 -97.55176969], avg: -95.66168070339587
XGBRegressor: scores: [-93.93518572 -96.92530663 -93.17217913 -92.95611203
-96.89379222], avg: -94.77651514481104
```

The following cell provides an evaluation of the performance of three ensemble algorithms of RandomForest and XGBoost. It takes about 5 hours to complete the cross-validation.

In [ ]:
```
elastic_net = ElasticNetCV()
ensemble_models = [VotingRegressor(estimators, n_jobs=-1),
                   StackingRegressor(estimators, cv=kf, n_jobs=-1),
                   StackingRegressor(estimators, elastic_net, cv=kf, n_jobs

for model in ensemble_models:
    scores = cross_val_score(model, features, targets, scoring='neg_mean_sc
    print(f'{type(model).__name__}: scores: {scores}, avg: {np.mean(scores)
```

```
VotingRegressor: scores: [-93.37383685 -95.81782547 -92.42927806 -92.145290
32 -95.89763872], avg: -93.93277388257981
StackingRegressor: scores: [-92.50578827 -95.03971448 -91.46993169 -91.0024
1121 -94.77617336], avg: -92.95880380298179
StackingRegressor: scores: [-93.12079008 -95.47860711 -92.09536035 -91.6146
7365 -95.42502812], avg: -93.54689186192984
```

Content-Based Recommendation System

In [ ]:
```
df = pd.read_csv('data.csv')
df2 = pd.read_csv('data_w_genres.csv')
```

Out[ ]:

	genres	artists	acousticness	danceability	duration_ms	energy	instrumenta
0	['show tunes']	"Cats" 1981 Original London Cast	0.590111	0.467222	250318.555556	0.394003	0.0
1	[]	"Cats" 1983 Broadway Cast	0.862538	0.441731	287280.000000	0.406808	0.0
2	[]	"Fiddler On The Roof" Motion Picture Chorus	0.856571	0.348286	328920.000000	0.286571	0.02
3	[]	"Fiddler On The Roof" Motion Picture Orchestra	0.884926	0.425074	262890.962963	0.245770	0.07
4	[]	"Joseph And The Amazing Technicolor Dreamcoat"...	0.510714	0.467143	270436.142857	0.488286	0.00

Data Preprocessing

In the following cell, I am going to:

- Extract special characters from the columns "genres" and "artists" of the dataset.
- Merge the two "regexed" artist features in the "artists_new"
- Define a new feature "artists_music" where a song and its artist name are concatenated to avoid dropping song with the same title, but from different artists.
- Drop duplicated songs in the new feature.

In [ ]:
```
df2['genres_new'] = df2['genres'].apply(lambda x: [re.sub(' ','_',i) for i
df['artists_2'] = df['artists'].apply(lambda x: re.findall(r"'([^']*)'", x)
df['artists_3'] = df['artists'].apply(lambda x: re.findall('\"(.*?)\"',x))
df['artists_new'] = np.where(df['artists_2'].apply(lambda x: not x), df['ar
df['artists_music'] = df.apply(lambda row: row['artists_new'][0]+row['name'
df.sort_values(['artists_music','release_date'], ascending = False, inplace
df.drop_duplicates('artists_music',inplace = True)
```

Next, I am going to:

- "explode" the "artists_new" feature to separate the multiple artists that might be authors of a song, according to "id".
- merge "artists_new" to "artists" and define "artists_ex_3" as the merged df where only not null values are stored.
- define "artgen" by grouping "artists_ex_3" on "id" according to "genres_new" and instantiate the column "genres_new" of "artgen" as a list of list of the genres.
- merge "clean_genre" on the df and fill null values with empty list.
- define "year" feature in df by extracting year from "release_date".
- create buckets of 5 points for popularity (as it might be difficult for an individual to actually sense the difference).
- define the "floats" encodable features.

In [ ]:
```
artists_ex = df[['artists_new','id']].explode('artists_new')
artists_ex_2 = artists_ex.merge(df2, how = 'left', left_on = 'artists_new',
artists_ex_3 = artists_ex_2[~artists_ex_2.genres_new.isnull()]
artgen = artists_ex_3.groupby('id')['genres_new'].apply(list).reset_index()
artgen['clean_genre'] = artgen['genres_new'].apply(lambda x: list(set(list(
df = df.merge(artgen[['id','clean_genre']], on = 'id',how = 'left')
df['clean_genre'] = df['clean_genre'].apply(lambda d: d if isinstance(d, li
df['year'] = df['release_date'].apply(lambda x: x.split('-')[0])
df['popularity'] = df['popularity'].apply(lambda x: int(x/5))
floats = df.dtypes[df.dtypes == 'float64'].index.values
```

Here I define the "encoder" function, it will be exploited in the next step to encode and give a name to the features created through TF-IDF.

In [ ]:
```
def encoder(df, col, name):
    enc_df = pd.get_dummies(df[col])
    feature = enc_df.columns
    enc_df.columns = [name + "|" + str(i) for i in feature]
    enc_df.reset_index(drop = True, inplace = True)
    return enc_df
```

In the following cell, I am going to:

- exploit tf-idf algorithm to encode "clean_genre" such that uncommon genres are weighted more when they appear, the intuition behind is that a very specific genre might be a relevant information to give recommendations.
- encode "year" and "popularity" through the predefined "encoder" function.
- scale through "MinMaxScaler" "floats".
- concatenate the encoded df.

In [ ]:
```
def vectorizer(df, floats):
    tfidf = TfidfVectorizer()
    tfidf_matrix =  tfidf.fit_transform(df['clean_genre'].apply(lambda x: '
    genre_df = pd.DataFrame(tfidf_matrix.toarray())
    genre_df.columns = ['genre' + "|" + i for i in tfidf.get_feature_names(
    genre_df.reset_index(drop = True, inplace=True)

    year_encoded = encoder(df, 'year','year') * 0.5
    popularity_encoded = encoder(df, 'popularity','pop') * 0.15

    floats = df[floats].reset_index(drop = True)
    scaler = MinMaxScaler()
    floats_scaled = pd.DataFrame(scaler.fit_transform(floats), columns = fl

    vectorized = pd.concat([genre_df, floats_scaled, popularity_encoded, ye
    vectorized['id']=df['id'].values
    return vectorized
```

In [ ]:
```
vecs = vectorizer(df, floats=floats)
```

In the following two cells, I am exploiting "spotipy" library to retrieve the information of the user playlists and their cover images. In particular:

- "client_id" and "client_secret" can be retrieved from https://developer.spotify.com/dashboard/login --> Login --> Create an App.
- To run the code effectively, the user has to run the code on a local machine (no Google Colab) and set "redirect_uri" attribute within the token variable, i.e., you can copy and paste (http://localhost:8881/callback=), then going back on the Spotify Dashboard --> Open the created app --> edit settings --> paste the url in "Redirect URIs" --> add --> save.

In [ ]:
```
client_id = 'd56a90db96a74e2282b55bd5266387fa'
client_secret = '5af77b47f633443c859e0397e6f8a522'
```

In [ ]:
```
scope = 'user-library-read'
auth_manager = SpotifyClientCredentials(client_id=client_id, client_secret=
sp = spotipy.Spotify(auth_manager=auth_manager)
token = util.prompt_for_user_token(scope, client_id= client_id, client_secr
sp = spotipy.Spotify(auth=token)

id_name = {}
list_photo = {}
for i in sp.current_user_playlists()['items']:
    id_name[i['name']] = i['uri'].split(':')[2]
    list_photo[i['uri'].split(':')[2]] = i['images'][0]['url']
```

Running the following cell, you should be able to see a list of your playlists.

In [ ]:
```
id_name
```

The following "playlist" function extracts the songs from a specific playlist given as input "playlist_name". In particular, it retrieves:

- Artist Name
- Track Name
- Track Id
- Track Album Image
- Date at which the song has been added to the playlist.

Finally, it sorts the playlist (dataframe) content according to "date_added".

In [ ]:
```
def playlist(playlist_name, id_dic, df):
    playlist = pd.DataFrame()
    playlist_name = playlist_name

    for ix, i in enumerate(sp.playlist(id_dic[playlist_name])['tracks']['it
        playlist.loc[ix, 'artist'] = i['track']['artists'][0]['name']
        playlist.loc[ix, 'name'] = i['track']['name']
        playlist.loc[ix, 'id'] = i['track']['id']
        playlist.loc[ix, 'url'] = i['track']['album']['images'][1]['url']
        playlist.loc[ix, 'date_added'] = i['added_at']

    playlist['date_added'] = pd.to_datetime(playlist['date_added'])
    playlist = playlist[playlist['id'].isin(df['id'].values)].sort_values('
    return playlist
```

The following "Recommender System" name has to be changed to the name of the chosen user's playlist.

In [ ]:
```
playlist = playlist('Recommender System', id_name, df)
```

The following "visualizer" function exploits the images retrieved in the previous "playlist" function to give a visualization of the album covers in a playlist. I will use this function both for the visualization of the songs actually contained in the playlist and the songs recommended by the algorithm.

In [ ]:

```
def visualizer(df):
    temp = df['url'].values
    plt.figure(figsize=(15,int(0.625 * len(temp))))
    columns = 5
    for i, url in enumerate(temp):
        plt.subplot(len(temp) / columns + 1, columns, i + 1)
        image = io.imread(url)
        plt.imshow(image)
        plt.xticks(color = 'w', fontsize = 0.1)
        plt.yticks(color = 'w', fontsize = 0.1)
        plt.xlabel(df['name'].values[i], fontsize = 12)
        plt.tight_layout(h_pad=0.8, w_pad=0)
        plt.subplots_adjust(wspace=None, hspace=None)
    plt.show()
```

In [ ]:

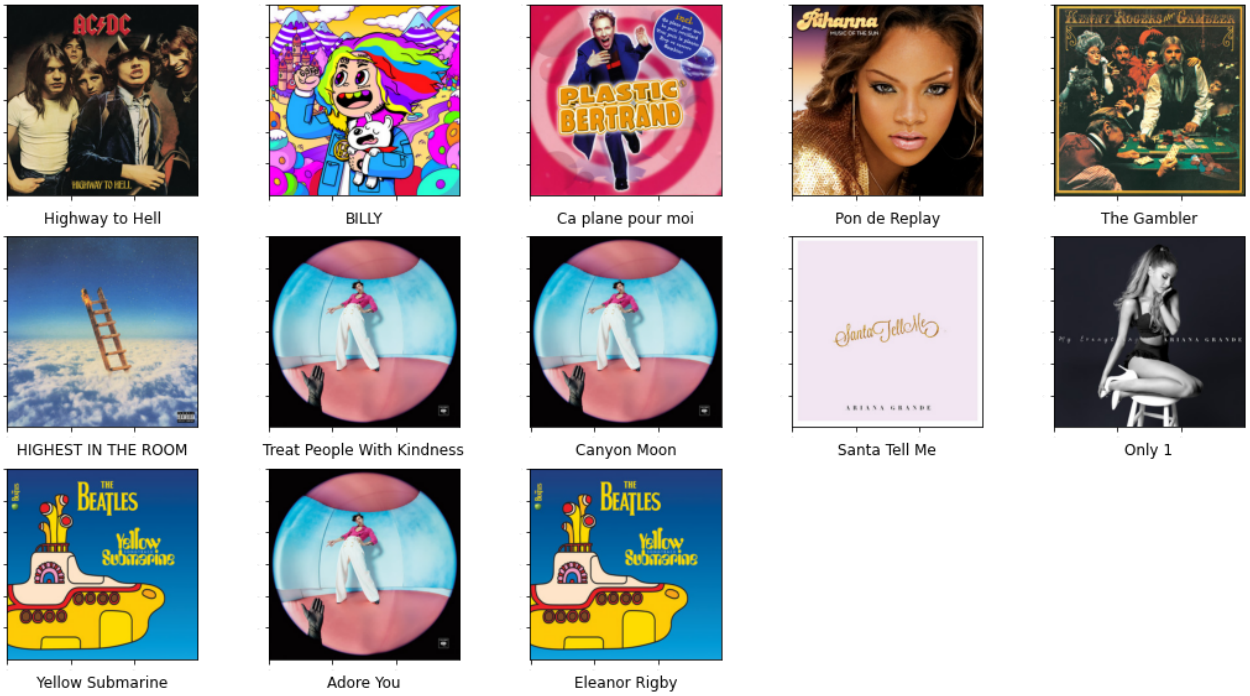
```
playlist
```

Out[ ]:

	artist	name	id	
24	AC/DC	Highway to Hell	2zYzyRzz6pRmhPzyfMEC8s	https://i.scdn.co/image/ab67616d00001e0
21	6ix9ine	BILLY	5KhFaq45chTw8RGfWo8T8J	https://i.scdn.co/image/ab67616d00001e0
15	Plastic Bertrand	Ca plane pour moi	71yCMlsD6qbD7NmNUEoVNR	https://i.scdn.co/image/ab67616d00001e0
14	Rihanna	Pon de Replay	4TsmezEQVSZNNPv5RJ65Ov	https://i.scdn.co/image/ab67616d00001e0
11	Kenny Rogers	The Gambler	5KqldkCunQ2rWxruMEtGh0	https://i.scdn.co/image/ab67616d00001e0
8	Travis Scott	HIGHEST IN THE ROOM	3eekarcy7kvN4yt5ZFzltW	https://i.scdn.co/image/ab67616d00001e0
6	Harry Styles	Treat People With Kindness	03mMSLEJCPoGJwQhHpN5y0	https://i.scdn.co/image/ab67616d00001e
5	Harry Styles	Canyon Moon	5IhzJOXNE7ki0IIJbZbnGq	https://i.scdn.co/image/ab67616d00001e
4	Ariana Grande	Santa Tell Me	0lizgQ7Qw35od7CYaoMBZb	https://i.scdn.co/image/ab67616d00001e0
3	Ariana Grande	Only 1	6LQzYkmd8ADbKOOEVDnlG4	https://i.scdn.co/image/ab67616d00001e0
2	The Beatles	Yellow Submarine	1tdltVUBkiBCW1C3yB4zyD	https://i.scdn.co/image/ab67616d00001e0
1	Harry Styles	Adore You	3jjujdWJ72nww5eGnfs2E7	https://i.scdn.co/image/ab67616d00001e
0	The Beatles	Eleanor Rigby	6EOKwO6WaLal58MSsi6U4W	https://i.scdn.co/image/ab67616d00001e0

In [ ]:

```
visualizer(playlist)
```



cThe following function represents the core of the recommender system implementation. Indeed, "featurizer" aims at giving a vectorial representation of the playlist such that it can be "mathematically" compared to other songs. In particular:

- It receives in input the playlist df and stores it in vecs_in after having sorted it according to "date_added".
- It defines vecs_out as the df to which the songs already in the playlist have been subtracted (i.e., we don't want to recommend songs already in the playlist).
- It then proceeds in evaluating the "months_elapsed" from when a song has been added to the playlist, indeed I will assign a weight based on this. (The intuition behind is that a playlist lasts for a longer time than our interests in certain songs and we might not update them so frequently, hence songs recently added might be a more relevant information of our tastes and directionate the recommendations better).
- Finally, to obtain the vector representing our playlist, each feature is multiplied by the predefined weight and column-wise summed to obtain the playlist weighted vector.

In [ ]:
```
def featurizer(vecs, playlist, weight):
    vecs_in = vecs[vecs['id'].isin(playlist['id'].values)].merge(playlist[[
    vecs_in = vecs_in.sort_values('date_added',ascending=False)
    vecs_out = vecs[~vecs['id'].isin(playlist['id'].values)]
    most_recent_date = vecs_in.iloc[0,-1]

    for ix, row in vecs_in.iterrows():
        vecs_in.loc[ix,'months_elapsed'] = int((most_recent_date.to_pydatet

    vecs_in['weight'] = vecs_in['months_elapsed'].apply(lambda x: weight **
    vecs_in.update(vecs_in.iloc[:,:-4].mul(vecs_in.weight,0))
    vecs_in = vecs_in.iloc[:, :-4]

    return vecs_in.sum(axis = 0), vecs_out
```

In [ ]:
```
vecs_in, vecs_out = featurizer(vecs, playlist, 1.09)
```

Finally, I define a "recommender" function which aims at comparing the playlist vector obtained through the previous function to the vector of the other songs. In order to do so, "Cosine Similarity" is exploited.

$$\text{cosine similarity} = S_C(A,B) := \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Cosine Similarity is of common-use both in Collaborative Filtering and Content-Based Recommender Systems. Intuitively, it computes the angle between two vectors and outputs a score in the range [-1, 1] with -1 meaning opposite directions and 1 meaning same direction. We now have a score over all of the songs in the dataset, we want to rank them for similarity with the playlist vector and extract a sample (top 15 in this case) of the ones with the highest score.

In [ ]:
```
def recommender(df, vecs_in, vecs_out):
    out = df[df['id'].isin(vecs_out['id'].values)]
    out['sim'] = cosine_similarity(vecs_out.drop('id', axis = 1).values, ve
    recommended = out.sort_values('sim',ascending = False).head(15)
    recommended['url'] = recommended['id'].apply(lambda x: sp.track(x)['alb
    return recommended
```

In [ ]:
```
recommended = recommender(df, vecs_in, vecs_out)
recommended
```

In [ ]:
```
visualizer(recommended)
```


bloodline
bad idea
Sunflower, Vol. 6
Paradise
thank u, next
in my head
NASA
I.F.L.Y.
Love Like That
Golden
Watermelon Sugar
fake smile
To Be So Lonely
imagine
make up