Spotify Analysis Report

July 11, 2021

Warner Music Group: Data Scientist, International Insights - Programming Exercise
Author: Jack Munday

Task: We would like you to connect programmatically to the public API of Spotify, get some interesting data and produce a little POC, a predictive analytics report or anything that you think worthwhile learning about a topic of music & audience of your choice. Feel free to use other data sources and any tools that you like.

I have used collected my data using Spotify's public API access through the Python SpotiPy library, performed a series of exploratory analyses of the data and then built a series of models to predict a song's popularity. My analysis is structured as follows:

1. Data Collections
2. Exploratory Analysis
3. Logistic Regression
4. Random Forest Classifier

The full set of source codes for this exercise can be found on my GitHub here.

Some of my other music-based projects can be found there too, which include:

- a Selenium-based web-scraper to automate tracking the historical prices of modern records in my wishlist; and
- a reconstruction of my Apple Music Replay statistics in BigQuery.

1 Data Collection

I have collected my data using the aforementioned SpotiPy Python library by building the `get_artist_data(artist_name, api_credentials)` function. For a given set of credentials to a Spotify developer application, this will search for the specified artist name and if a match is found download all songs of that artist's album. The full docstring for this is included below in the function definition. I have then iterated through a list of artist names obtained from my personal Apple Music library to download a sufficiently large body of data to draw some insightful conclusions.

[ ]:
```
%pip install spotipy --quiet
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
%cd /content/gdrive/MyDrive/spotify/scripts
import numpy as np
import pandas as pd
from spotipy.oauth2 import SpotifyClientCredentials
```

```
import spotipy
# Custom library containing spotify credentials for authentication
import spotify_credentials as cred
import os
import glob
import time
import random
```

```
Mounted at /content/gdrive
/content/gdrive/MyDrive/spotify/scripts
```

[ ]:
```
def get_artist_data(artist_name, api_credentials):
    """
    Function that calls the Spotify API using Pythons SpotiPy library to search
    for a specified artist_name within Spotify's dataset. If a match is found
    that artist / bands albums will be appended into a nested dictionary along
    with each album's subsequent tracks and audio features as classified by
    Spotify. More information on the meaning of each feature can be found at
    https://developer.spotify.com/documentation/web-api/reference/
    #category-tracks.

    inputs:
    ---------------------------------------------------------------------------
    artist_name:        Artist name whose catelog is to be downloaded
    api_crediantials:   Credentials required to call Spotify API. i.e. output of
                        calling SpotifyClientCredentials().

    returns:
    ---------------------------------------------------------------------------
    Unique dataframe for each artist_name which contains their whole Spotify
    catelog, with a series of categorisation features as described in the
    category-tracks url above.
    """

    sp = spotipy.Spotify(client_credentials_manager=api_credentials, retries=15)
    # Search for artist name, find all their uris (unique reference ids) and
    # the corresponding album names storing: storing both in seperate lists.
    search_result = sp.search(artist_name)
    artist_uri = search_result['tracks']['items'][0]['artists'][0]['uri']
    # Top artist name search results
    artist_name_search_result =␣
↪search_result['tracks']['items'][0]['artists'][0]['name']

    # If the search result doesn't match the input artist name,
    # then look through the top 10 results, if still no match skip this artist.
    if artist_name != artist_name_search_result:
        try:
```

```
            top_10_results =␣
↪[search_result['tracks']['items'][i]['artists'][0]['name']
                                for i in range(10)]
            # Get index position of matched artist name in list to use as an
            # index-match in artist_name_search_result.
            index = top_10_results.index(artist_name)
            artist_uri =␣
↪search_result['tracks']['items'][index]['artists'][0]['uri']
            artist_name_search_result =␣
↪search_result['tracks']['items'][index]['artists'][0]['name']
        except:
            print(f"!! {artist_name} not found in Spotify dataset.")
            return 0

    sp_albums = sp.artist_albums(artist_uri, album_type='album')

    album_names = [sp_albums['items'][i]['name']
                   for i in range(len(sp_albums['items']))]
    album_uris = [sp_albums['items'][i]['uri']
                  for i in range(len(sp_albums['items']))]

    print(f">> Currently downloading {artist_name_search_result}'s data.")

    ################################################################################
    # GET TRACK NAMES, SEQUENCING & IDS FOR EACH ARTIST ALBUM
    ################################################################################
    spotify_albums = {}
    album_counter = 0
    track_keys = ['artist_name', 'album', 'track_number', 'id', 'name', 'uri']

    for album in album_uris:
        # Assign an empty list to each key value inside a nested dictionary.
        spotify_albums[album] = {key: [] for key in track_keys}

        # Pull track data for each album track and append its info to nest dict.
        tracks = sp.album_tracks(album)

        for n in range(len(tracks['items'])):
            spotify_albums[album]['artist_name'].
↪append(artist_name_search_result)
            spotify_albums[album]['album'].append(album_names[album_counter])
            spotify_albums[album]['track_number'].
↪append(tracks['items'][n]['track_number'])
            spotify_albums[album]['id'].append(tracks['items'][n]['id'])
            spotify_albums[album]['name'].append(tracks['items'][n]['name'])
            spotify_albums[album]['uri'].append(tracks['items'][n]['uri'])
```

```
        album_counter += 1

    ###############################################################################
    # GET AUDIO FEATURES FOR EACH ALBUM TRACK
    ###############################################################################
    audio_feature_keys = ['acousticness', 'danceability', 'energy',
                          'instrumentalness', 'liveness', 'loudness',
                          'speechiness', 'tempo', 'valence', 'duration_ms',
                          'release_date', 'popularity']
    for album in spotify_albums:
        # Assign audio feature keys empty list values in nested dictionary.
        for key in audio_feature_keys:
            spotify_albums[album][key] = []

        for track in spotify_albums[album]['uri']:
            # Get all audio features for the current track and append values
            # into appropriate key in dictionary.
            features = sp.audio_features(track)

            # Append data for all keys expect duration, release date and
            # popularity (final three elements in audio_feature_keys) which
            # will need to be obtained using sp.track().
            for key in audio_feature_keys[:-3]:
                spotify_albums[album][key].append(features[0][key])

            track_info = sp.track(track)

            spotify_albums[album]['duration_ms'].
↪append(track_info['duration_ms'])
            spotify_albums[album]['release_date'].
↪append(track_info['album']['release_date'])
            spotify_albums[album]['popularity'].append(track_info['popularity'])

    ###############################################################################
    # REORGANISE DATA INTO AN UNNESTED DICTIONARY TO ALLOW FOR DF CONVERSION
    ###############################################################################
    all_albums_data_keys = track_keys + audio_feature_keys
    all_albums_data = {key: [] for key in all_albums_data_keys}

    for album in spotify_albums:
        for feature in spotify_albums[album]:
            all_albums_data[feature].extend(spotify_albums[album][feature])

    df = pd.DataFrame.from_dict(all_albums_data)
    return df
```

1.1 Data Download Pipeline

The `main` function below runs a data processing pipeline that calls `get_artists_data()` to collect a specified artist name's data using Spotify's API via the SpotiPy python library.

1.1.1 Importing credentials

To authenticate this process, I have written a basic library (`spotify_credentials`), which contains the credentials to verify access to my Spotify Developer account. Separating crediants from the main script allows the user to keep their credentials private when uploading to Git, while also avoiding the need to continually export their client id and secret as environment variables every time the script is run.

To replicate save your credentials under the following `spotify_credentials.py` in `scripts/` as follows:

```
client_id = "xxx"
client_secret = "xxx"
redirect_url = "http://localhost:8888
```

A client id can then be accessed by calling `spotify_crediantial.client_id`.

1.1.2 Generating a list of artists name's to collect data

As I am not a user of Spotify, I have generated a list of artists from my Apple Music library, from a prior data & privacy request submitted to Apple. While this was not necessary for the analysis - I could have easily generated the list from another source - I already had the data I thought it would be a nice touch to have a dataset that is personal to my tastes. This has given me a list of 1,084 unique artists for which I have downloaded each of their whole music catelogues - resulting in an output dataset of around 70k songs. I consider myself to have a broad taste in music, but this dataset will naturally contain a skew, if this proves an issue I will combine data from additional sources to balance out my dataset.

1.1.3 Parallelising Data Download

Since this is a one-time request for data, there would be little time-cost benefit to efficiently parallelising my code. Although I have randomly shuffled the artist name input list on each run of the python script, which has allowed me to run multiple threads of my script at the same time, without each script iterating over the same part of the input list. The speed-up gained by this is significantly outweighed by the cost of checking if an artist name has already been processed on each iteration. Saving each artist's data as a seperating file and the merging on completion also allows me to checkpoint my code, in the sense that if the https times out or Spotify forcibly disconnects me I can easily pick up where I left off.

[ ]:
```
def main():

    credentials = SpotifyClientCredentials(client_id=cred.client_id,
                                           client_secret=cred.client_secret)
    ###############################################################################
    # GET ARISTS IN MY APPLE MUSIC LIBRARY
    ###############################################################################
```

```
    # If artist list has not already been generated, read in apple music library
    # data and drop all cols expect album artist.
    if not os.path.isfile("../data/artist_list.csv"):
        in_dir = "~/Documents/Computing/SQL/apple_music_replay/input_data/
↪MusicLib.csv"
        artists_df = pd.read_csv(in_dir, usecols=["Album Artist"])
        artists_df.drop_duplicates(inplace=True)
        artists_df.sort_values(by=['Album Artist'], inplace=True,␣
↪ascending=True)
        artists_df.to_csv("../data/artist_list.csv", index=False)
    else:
        artists_df = pd.read_csv("../data/artist_list.csv")
    artists_list = artists_df.values.tolist()

    # Randomly shuffle artist_list to allow for multiple processors to be run
    # the script simultaneously. This parallelisation more than accounts for the
    # slow down in having to check whether a dataframe for the arists has
    # already been download on each iteration, without having to deploy any
    # libraries to parallelise my code.
    random.shuffle(artists_list)

    ###############################################################################
    # CALL get_artist_data() FOR EACH ARTIST IN artists_list
    ###############################################################################
    request_counter = 0
    sleep_min, sleep_max = 4, 6
    for artist in artists_list:
        # If artist dataframe doesn't exist then call get_artist_data() to
        # download artist's data.
        if not os.path.isfile('../data/artists/' + str(*artist) + '.csv'):
            df = get_artist_data(*artist, credentials)
            request_counter += 1
            # Add random delay to avoid being forcibly disconnected.
            if request_counter % 5 == 0:
                time.sleep(np.random.uniform(sleep_min, sleep_max))
            # Get artist data returns 0 on not an unmatched artist, else a
            # pandas dataframe for matched artists. Check that we do not have
            # our error code (0) before trying to write df to disk.
            if type(df) != int:
                df.to_csv('../data/artists/' + str(*artist) + '.csv',
                          index=False)

    ###############################################################################
    # APPEND ARTISTS DATAFRAMES INTO MASTER DATAFRAME
    ###############################################################################
    # Create an empty master dataframe to append each artists catelog to.
    master_df = pd.DataFrame()
```

```
    artist_csvs = glob.glob(os.path.join("../data/artists/","*.csv"))
    for f in artist_csvs:
        df = pd.read_csv(f)
        master_df = master_df.append(df, ignore_index=True)

    master_df.to_csv('../data/master_data.csv', index=False)
    return 1
```

[ ]:
```
if __name__ == "__main__":
  main()
```

I ran this script locally - not on Google Colab - as it allowed me to simultaneously run many threads to speed up the time required to download the dataset. Consequently, all cells in this notebook have no output, I have presented this portion of my analysis in Google Colab for continuity with my other notebooks, where using a Juypter Notebook works better for displaying graphs, data-frames etc than a terminal console.

2 Expoloratory Analysis of Dataset

This notebook performs an exploratory analysis of the Spotify dataset that has just been generated by calling the Spotify API through SpotiPy. It has allowed me to gain a more detailed understanding of what each of the audio features means before moving on to using this dataset to build predictive models.

The analysis is laid out over the following sections:

- 2.1 Investigating Dataset's Structure
- 2.2 Audio Feature Distributions
- 2.3 Song Feature Yearly Trends
- 2.4 Correlation of Audio Features
- 2.5 Correlation of Popularity with Audio Features

[1]:
```
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
%cd /content/gdrive/MyDrive/spotify/scripts
pd.set_option('display.max_rows', 10)
```

```
Mounted at /content/gdrive
/content/gdrive/MyDrive/spotify/scripts
```

2.1 Investigating Dataset's Structure

To understand what each of the audio features means beyond that stated in Spotify's documentation, I have begun by filtering the dataset for an album I know very well.

[9]:
```
df = pd.read_csv("../data/master_data.csv")
df['year'] = df['release_date'].apply(lambda x: int(x[:4]))
print(f"SHAPE: {df.shape}")
print(f"COLUMNS: {list(df.columns)}")
df[df["album"] == "What Kinda Music"].head(12)
```

```
SHAPE: (71292, 19)
COLUMNS: ['artist_name', 'album', 'track_number', 'id', 'name', 'uri',
'acousticness', 'danceability', 'energy', 'instrumentalness', 'liveness',
'loudness', 'speechiness', 'tempo', 'valence', 'duration_ms', 'release_date',
'popularity', 'year']
```

[9]:

	artist_name	album	…	popularity	year
46760	Yussef Dayes	What Kinda Music	…	56	2020
46761	Yussef Dayes	What Kinda Music	…	51	2020
46762	Yussef Dayes	What Kinda Music	…	61	2020
46763	Yussef Dayes	What Kinda Music	…	56	2020
46764	Yussef Dayes	What Kinda Music	…	45	2020
…	…	…	…	…	…
46767	Yussef Dayes	What Kinda Music	…	47	2020
46768	Yussef Dayes	What Kinda Music	…	54	2020
46769	Yussef Dayes	What Kinda Music	…	48	2020
46770	Yussef Dayes	What Kinda Music	…	44	2020
46771	Yussef Dayes	What Kinda Music	…	45	2020

[12 rows x 19 columns]

[10]:
```
df.describe()
```

[10]:

	track_number	acousticness	…	popularity	year
count	71292.000000	71292.000000	…	71292.000000	71292.00000
mean	7.881389	0.311502	…	30.080766	2010.39414
std	5.672300	0.314018	…	15.218917	12.06896
min	1.000000	0.000001	…	11.000000	1961.00000
25%	4.000000	0.038300	…	18.000000	2007.00000
50%	7.000000	0.185000	…	27.000000	2015.00000
75%	11.000000	0.547000	…	39.000000	2019.00000
max	50.000000	0.996000	…	97.000000	2021.00000

[8 rows x 13 columns]

2.2 Audio Feature Distributions

Now let's see what the most popular songs are in the dataset that I have collected. Please note that the dataset is formed from artists that are in my apple music library, so it is not reflective of the true most popular songs on Spotify.

I decided to take this route to add some personalisation to the analysis in lieu of the fact that I do not have any Spotify streaming data which could be analysed. As this is just a POC, this is not too

much of a concern. In production, I can simply run the same `get_spotify_data()` function for a distribution of artist names that is reflective of the overall industry and not my personal taste.

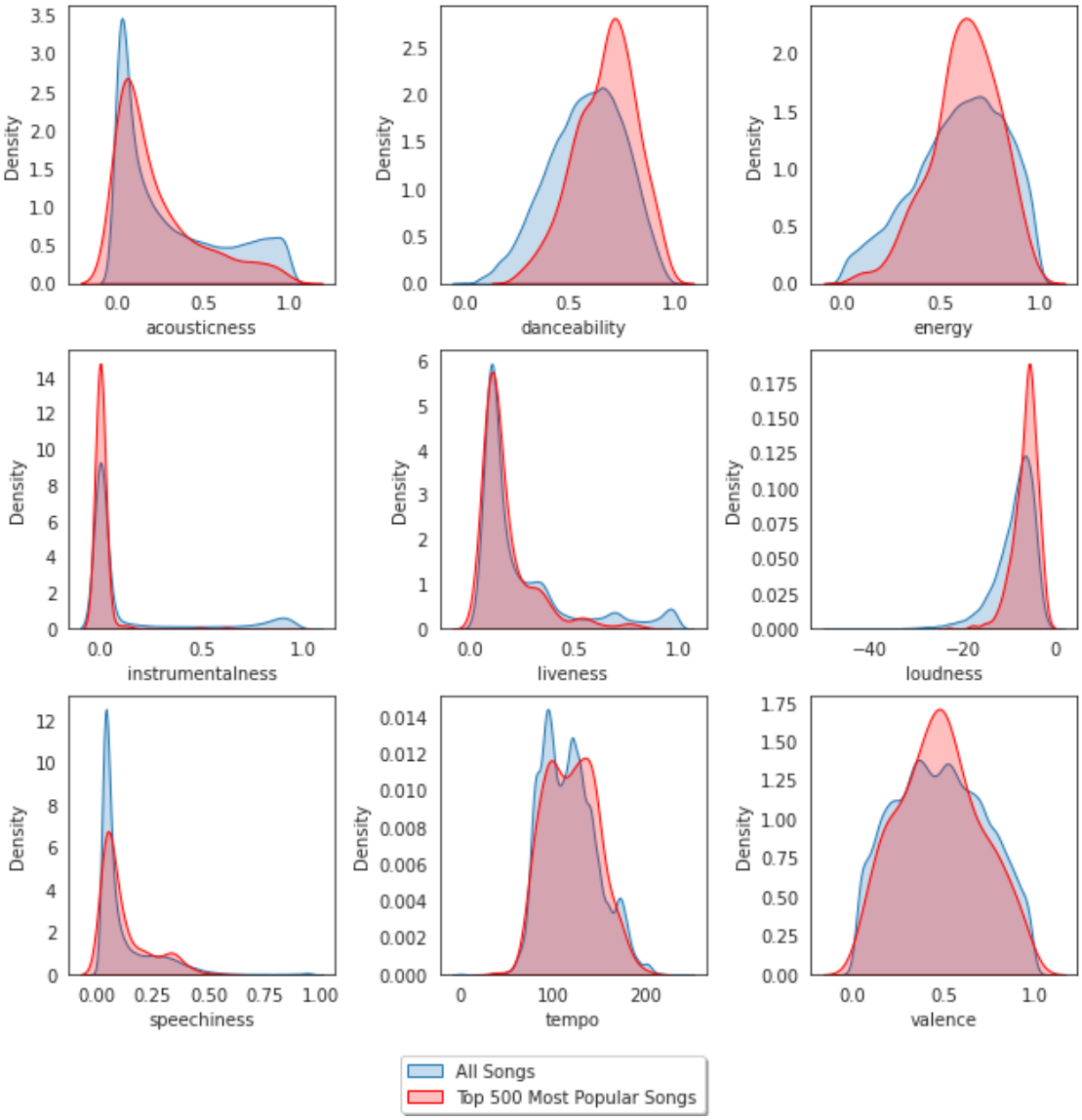
[11]:
```
%pip install pandasql --quiet
import pandasql as ps
query = """
        SELECT artist_name, name `track name`, popularity
        FROM df
        ORDER BY popularity DESC
        LIMIT 10
        """
print(ps.sqldf(query, locals()))

# Top 500 songs in dataset my popularity, identical to the SQL query
# above. I have used an SQL query above as it's far easier to drop
# any columns that I'm not iterested in printing.
top_500 = df.sort_values(by="popularity", ascending=False).head(500)
```

```
     artist_name                                track name  popularity
0   Justin Bieber  Peaches (feat. Daniel Caesar & Giveon)          97
1      The Weeknd                          Blinding Lights          94
2        Dua Lipa             Levitating (feat. DaBaby)          92
3      The Weeknd                          Save Your Tears          91
4        Doja Cat                                 Streets          89
5   Lewis Capaldi                     Someone You Loved          88
6     Post Malone                                 Circles          88
7           DaBaby          ROCKSTAR (feat. Roddy Ricch)          88
8   Ariana Grande                               positions          88
9        Dua Lipa                              We're Good          87
```

To understand what makes a song popular I have plotted the distribution for each audio feature for all songs in the dataset (blue) and the top 500 songs in the dataset (red). The majority of features show a slight distribution shift when comparing 'all songs' to the 'top 500'. Instrumentallness shows the lowest deviation between the two datasets implying that this feature will likely be of the lowest predictive power when determining whether or not a song will be popular.

[12]:
```
sns.set_style("white")
audio_features = ['acousticness', 'danceability', 'energy', 'instrumentalness',
       'liveness', 'loudness', 'speechiness', 'tempo', 'valence']

plt.rcParams['figure.figsize'] = (9, 9)
fig, axes = plt.subplots(3,3)

sns.kdeplot(df[audio_features[0]], shade=True, ax=axes[0][0])
sns.kdeplot(df[audio_features[1]], shade=True, ax=axes[0][1])
sns.kdeplot(df[audio_features[2]], shade=True, ax=axes[0][2])
sns.kdeplot(df[audio_features[3]], shade=True, ax=axes[1][0])
sns.kdeplot(df[audio_features[4]], shade=True, ax=axes[1][1])
```

```
sns.kdeplot(df[audio_features[5]], shade=True, ax=axes[1][2])
sns.kdeplot(df[audio_features[6]], shade=True, ax=axes[2][0])
sns.kdeplot(df[audio_features[7]], shade=True, ax=axes[2][1])
sns.kdeplot(df[audio_features[8]], shade=True, ax=axes[2][2])

sns.kdeplot(top_500[audio_features[0]], color="red", shade=True, ax=axes[0][0])
sns.kdeplot(top_500[audio_features[1]], color="red", shade=True, ax=axes[0][1])
sns.kdeplot(top_500[audio_features[2]], color="red", shade=True, ax=axes[0][2])
sns.kdeplot(top_500[audio_features[3]], color="red", shade=True, ax=axes[1][0])
sns.kdeplot(top_500[audio_features[4]], color="red", shade=True, ax=axes[1][1])
sns.kdeplot(top_500[audio_features[5]], color="red", shade=True, ax=axes[1][2])
sns.kdeplot(top_500[audio_features[6]], color="red", shade=True, ax=axes[2][0])
sns.kdeplot(top_500[audio_features[7]], color="red", shade=True, ax=axes[2][1])
sns.kdeplot(top_500[audio_features[8]], color="red", shade=True, ax=axes[2][2])

legend = fig.legend(['All Songs', 'Top 500 Most Popular Songs'],
                    loc="lower center", fontsize="medium",
                    borderaxespad=0.2, bbox_to_anchor=(0.5, 0), shadow=True)

bbox = legend.get_window_extent(fig.canvas.get_renderer()).transformed(fig.
 ↪transFigure.inverted())
fig.tight_layout(rect=(0, bbox.y1, 1, 1), h_pad=0.5, w_pad=0.5)

plt.show()
```



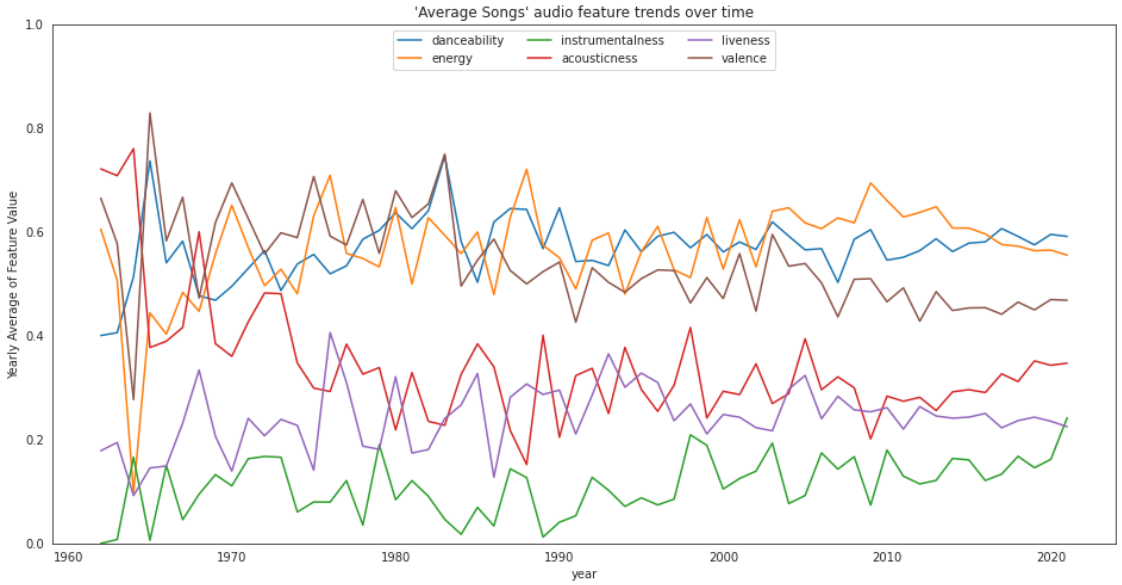
2.3 Song Feature Yearly Trends

Some of the high-level trends in the average of these audio features can be attributed to industry-wide movements. The trends are significantly more noisy in the earlier years due to the lower number of songs in these years in my dataset. The dataset I have generated based on artists in my Apple Music Libraryi, which is skewed to more recent artists results in a smoother averaging after the late 90s.

Below I am reducing each year down to essentially the "average song" for that year and plotting the yearly variation across these audio features, which is clearly not reflective of the breadth and depth of music released in that year. However, we can clearly see key macro trends which correlate I would attribute various genre movements:

- Acousticness of the average song drops from the '60s to a minimum in the '90s, where it seems to have now reached a plateau. This is clearly explained by the increased dominance of electric instrumentation which reached peak saturation in the late '90s with the rise of modern dance music and EDM.
- Valence, danceability and energy (which we would expect high scores for all three for disco music) rise from '60s reach a peak in the '80s at the height of the disco movement before plateauing out. i.e. Disco truly is not dead.
- While there is significant noise in this trend, it appears that instrumental music is on the rise with the growing popularity of lo-fi hip hop and ambient music. A slight but sharp increase is observed in the instrumentallness trendline. It is important to note that this peak is massively suppressed in the averaging operation due to the fact that instrumental music makes up such a small portion of the overall music catelogue available.

[16]:
```
from sklearn.preprocessing import MinMaxScaler

yearly_features = df[[*audio_features, "year"]].groupby("year").mean().
 ↪sort_values(by="year").reset_index()

# Rescale loudness and tempo to be bounded between 0 & 1.
scaler = MinMaxScaler()
yearly_features["loudness"] = scaler.fit_transform(yearly_features["loudness"].
 ↪values.reshape(-1,1))
yearly_features["tempo"] = scaler.fit_transform(yearly_features["tempo"].values.
 ↪reshape(-1,1))

# Let's focus on a few key features to keep the plot legible.
key_feats = ['danceability', 'energy', 'instrumentalness', 'acousticness',
             'liveness', 'valence']

yearly_features = yearly_features[["year", *key_feats]]
yearly_features_melt = yearly_features.melt(id_vars="year",
                                            var_name="Audio Feature",
                                            value_name="Yearly Average of␣
 ↪Feature Value")
plt.figure(figsize=(16,8))
plt.title("'Average Songs' audio feature trends over time")
sns.lineplot(x="year", y="Yearly Average of Feature Value",
             hue="Audio Feature", data=yearly_features_melt)
plt.legend(loc="upper center", ncol=3)
plt.ylim([0,1])
plt.show()
```

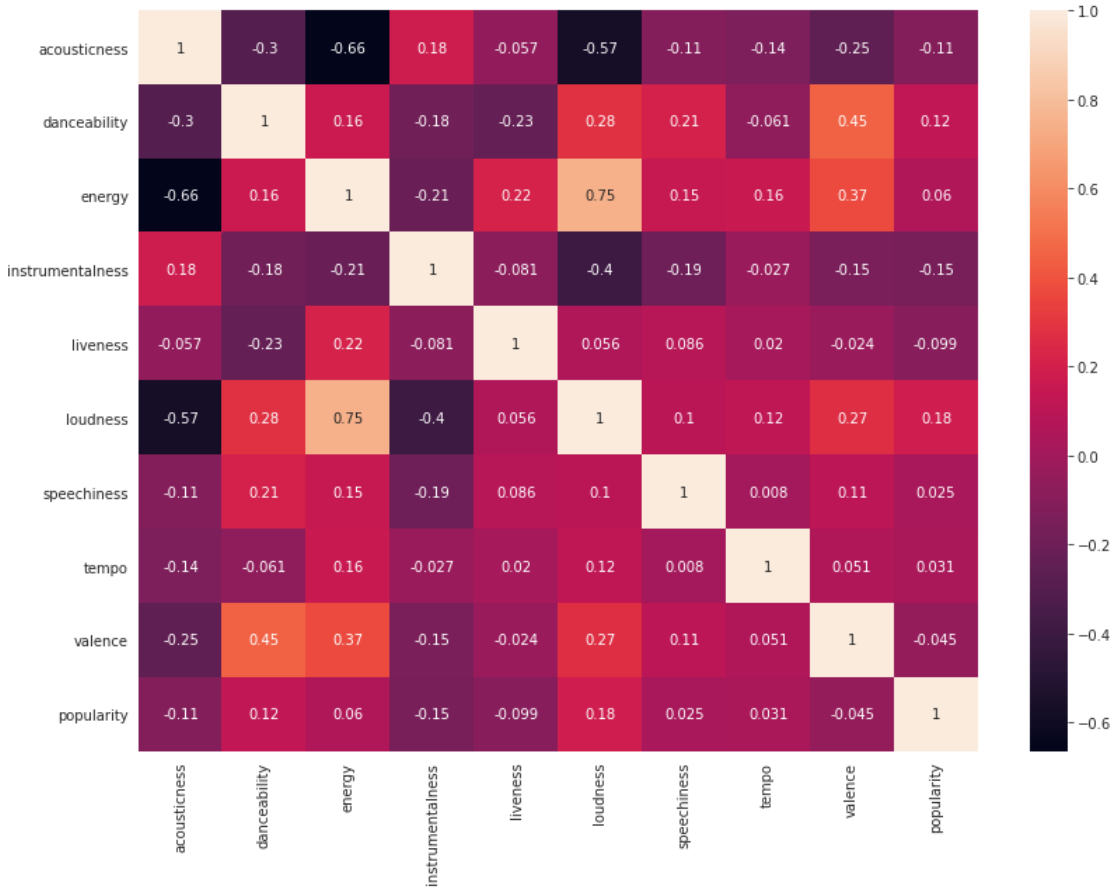


2.4 Correlation of Audio Features

Below plots, the correlation of all audio features against each other. While this doesn't derive any particular insight into the correlation of a specific audio feature to popularity, this is to be expected due to the complexities of music and the breadth of genres and styles. There is no single feature of music that correlates with popularity. However, this was helpful to confirm some of my intuitive expectations behind the meaning of each audio feature, I have assumed a few of the strongest correlations between audio features below:

- Acousticness is strongly negatively correlated with energy and valence.
- Loudness has a strong positive correlation with energy.
- Danceability and valency are positively correlated, similarly valence is positively correlated with energy.
- I initialled expected that acousticness would show a very strong positive correlation with instrumentalness. This is clearly an incorrect preconception - as there is no reason why instrumental music can not use electronic instrumentation - this reflects more so on my listening habits in instrumental music.

[14]:
```
corr = df[[*audio_features, "popularity"]].corr()
plt.figure(figsize=(12, 9))
sns.heatmap(corr, annot=True)
plt.tight_layout()
```



2.5 Correlation of Popularity with Audio Features

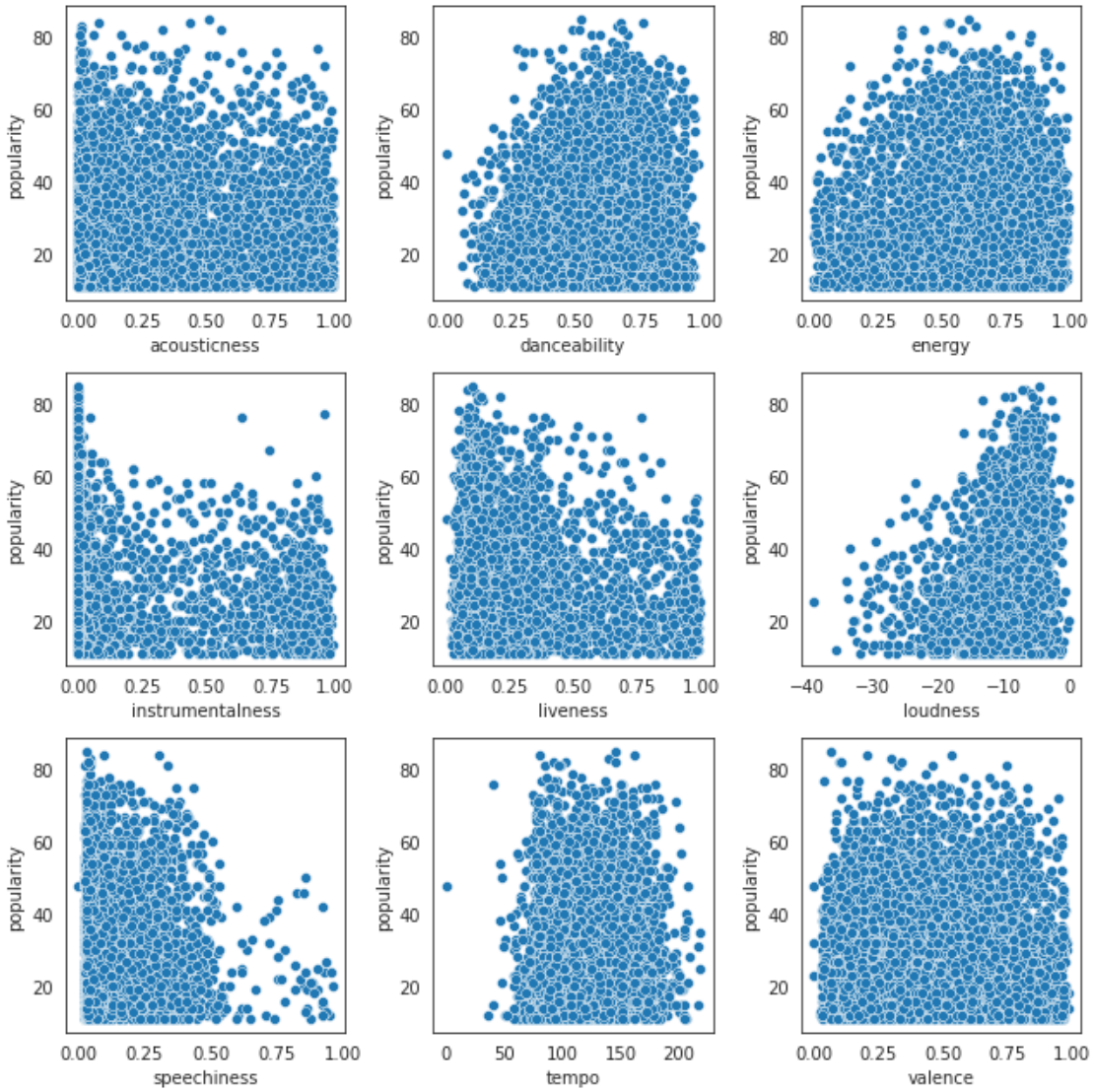
To emphasis the lack of correlation between popularity and all the features, I plot have produced scatter plots for each audio feature.

Each feature is largely uniformly distributed, while a few audio features show some key trends:

- Instrumentalness show two bands of clusters around 0 and 1, as to expected because a song is typically either instrumental or not.
- Songs typically have a speechiness of > 0.5.
- Unsuprinsingly, tempo is generally confined between 50 and 200 BPM.

[17]:
```
# Plot scatter plots of each variable against popularity to identify any macro
# trends.
# Get a random sample of 5000 rows to speed up plotting
df = df.sample(n=5000)

fig, axes = plt.subplots(3,3)

sns.scatterplot(x=audio_features[0], y="popularity", data=df, ax=axes[0][0])
```

```
sns.scatterplot(x=audio_features[1], y="popularity", data=df, ax=axes[0][1])
sns.scatterplot(x=audio_features[2], y="popularity", data=df, ax=axes[0][2])

sns.scatterplot(x=audio_features[3], y="popularity", data=df, ax=axes[1][0])
sns.scatterplot(x=audio_features[4], y="popularity", data=df, ax=axes[1][1])
sns.scatterplot(x=audio_features[5], y="popularity", data=df, ax=axes[1][2])

sns.scatterplot(x=audio_features[6], y="popularity", data=df, ax=axes[2][0])
sns.scatterplot(x=audio_features[7], y="popularity", data=df, ax=axes[2][1])
sns.scatterplot(x=audio_features[8], y="popularity", data=df, ax=axes[2][2])

plt.tight_layout()
plt.show()
```



I had initially intended to build a series of machine learning models from simple to complex (i.e. Linear Regression up to a Neural Net) that would predict a song's popularity score. From the above, it is clear that a linear regression model will significantly underfit the dataset. Instead, I begin by building a logistic regression model to predict whether a given song is a hit or not (popularity > 50) and then moving on to more complex algorithms such as boosted random forest, XG Boost and neural networks.

3 Logistic Regression

This notebook builds a logitstic regression model to predict whether or not a song will be a hit or not based on its audio features as classified by Spotify (e.g. duration_ms, danceability, liveness etc.). A song is defined as successful if it achieves a minimum popularity score of 50. All data has been collected using Spotify's API as accessed through python's SpotiPy library - this is fully detailed in the prior notebook using the script defined in `scripst/get_spotify_data.py`.

Time permitting, I hope to extend the accuracy of this initial classification using Billboard's API, by defining a song as a hit if it appears in their top / hot 100. As I am aware that defining success based purely on Spotify's popularity classification is not a fair measure of a song's success.

This notebook is divided out as follows:

- 2.1 Feature Engineering
- 2.2 Logistic Regression Model
- 2.3 Hyperparameter Tuning
- 2.4 Final Model

[1]:
```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
from google.colab import drive
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, precision_score, f1_score
from sklearn.metrics import classification_report
%pip install vaderSentiment  --quiet
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
drive.mount('/content/gdrive', force_remount=True)
%cd /content/gdrive/MyDrive/spotify/scripts/
```

```
     |                | 133kB 7.7MB/s
Mounted at /content/gdrive
/content/gdrive/MyDrive/spotify/scripts
```

3.1 Feature Engineering

I have begun by creating a series of features to help my model with its classification:

- `hit?` - If a song has a `popularity > 50` it is defined as a hit (1) else 0.
- `year` - Stripping `release_date` to just the year of release.

- `sentiment` - Sentiment of `name` as classified using VADER.

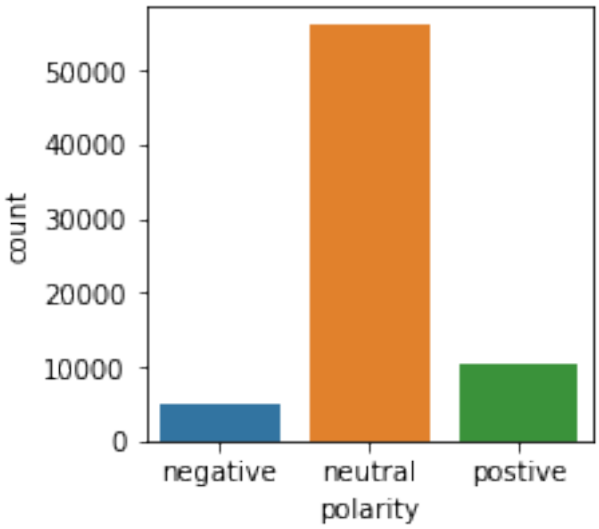
I have then checked for any NaN values and replaced these with the mean value of that column while dropping any non-numeric features.

[2]:
```
df = pd.read_csv("../data/master_data.csv")
audio_features = ['acousticness', 'danceability', 'energy', 'instrumentalness',
       'liveness', 'loudness', 'speechiness', 'tempo', 'valence']
hit_cutoff = 50
df['hit?'] = df['popularity'].apply(lambda x: 1 if x>=hit_cutoff else 0)
print(df['hit?'].value_counts(normalize=True))

df['year'] = df['release_date'].apply(lambda x: int(x[:4]))
# Sentiment analysis of track title using VADER
analyser = SentimentIntensityAnalyzer()
df['sentiment'] = df['name'].apply(lambda x:
                                   analyser.polarity_scores(x)['compound'])
```

```
0    0.872103
1    0.127897
Name: hit?, dtype: float64
```

We see here that our dataset has ended up being unbalanced, with ~13% of songs defined as hits (popularity > 50). This will need to be accounted for in training by weighting our classes to prevent the model from underfitting to the dataset.

[3]:
```
cut_bins = [-1, -0.33, 0.33, 1]
cut_labels = ["negative", "neutral", "postive"]

sentiment_df = pd.DataFrame()
sentiment_df['polarity'] = pd.cut(df['sentiment'], bins=cut_bins,
                                  labels=cut_labels)

plt.figure(figsize=(3,3))
sns.countplot(x='polarity', data=sentiment_df)
plt.show()
```



Due to the generally short song names and the lack of any context, most songs have been assigned a 'neutral' sentiment.

[4]:
```
#replace duartion NaNs with the mean column value.
duration_mean = df['duration_ms'].mean()
df['duration_ms'].fillna(duration_mean, inplace=True)
# Check for nulls or NaNs before training
print(pd.isnull(df).sum())
```

```
artist_name         0
album               0
track_number        0
id                  0
name                0
uri                 0
acousticness        0
danceability        0
energy              0
instrumentalness    0
liveness            0
loudness            0
speechiness         0
tempo               0
valence             0
duration_ms         0
release_date        0
popularity          0
hit?                0
year                0
sentiment           0
```

```
dtype: int64
```

To speed up the convergence of my algorithm, I have normalised all my input features. This will become particularly important later on when building neural networks to prevent exploding / vanishing gradients.

[5]:
```
# Drop text features and normalise the remaining numeric features.
df.drop(labels=["track_number", "artist_name", "year", "album", "name", "id",
                "uri", "release_date"], axis=1, inplace=True)

from sklearn.preprocessing import MinMaxScaler
x = df.values
min_max_scaler = MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(x)
df = pd.DataFrame(x_scaled, columns = [*audio_features, "duration_ms",
                                       "popularity", "hit?", "sentiment"])
print(df.columns)
```

```
Index(['acousticness', 'danceability', 'energy', 'instrumentalness',
       'liveness', 'loudness', 'speechiness', 'tempo', 'valence',
       'duration_ms', 'popularity', 'hit?', 'sentiment'],
      dtype='object')
```

3.2 Logistic Regression Model

To begin training our model, we first must split the data into test and train sets. I have then defined a helper function to plot the confusion matrix given a set of predictions and actual values.

[6]:
```
x = df.drop(["hit?", "popularity"], axis=1)
y = df.loc[:,"hit?"].values

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
df.head()
```

[6]:

	acousticness	danceability	energy	…	popularity	hit?	sentiment
0	0.338353	0.763931	0.352339	…	0.069767	0.0	0.492009
1	0.188754	0.660588	0.472462	…	0.046512	0.0	0.492009
2	0.652610	0.601824	0.494484	…	0.069767	0.0	0.492009
3	0.915663	0.270517	0.177161	…	0.034884	0.0	0.492009
4	0.312248	0.746707	0.295281	…	0.046512	0.0	0.534119

[5 rows x 13 columns]

[7]:
```
def conf_matrix(y_test, pred_test):
    con_mat = confusion_matrix(y_test, pred_test)
    con_mat = pd.DataFrame(con_mat, range(2), range(2))

    plt.figure(figsize=(3,3))
    sns.set(font_scale=1)
```

```
    sns.heatmap(con_mat, annot=True, annot_kws={"size": 12}, fmt='g',
                cmap='Blues', cbar=False)
    plt.ylabel("Actual Values")
    plt.xlabel("Predicted Values")
    plt.show()
```

3.2.1 Class Imbalance

When training our algorithm, the severe class imbalance needs to be accounted for to create a model that properly fits the data. Using accuracy would not be a suitable metric to define performance, as naively the model could achieve around 87% accuracy just by classifying everything as not being not a hit. Sci-kit learn has a number of default class weightings:

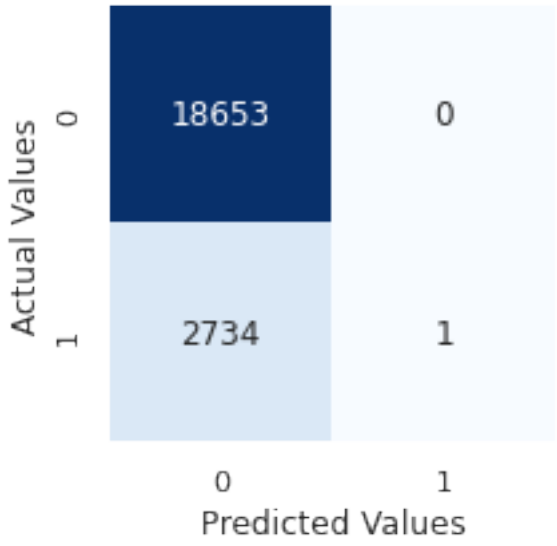
`"None"` - Each class is assigned an equal weighting, in the case of a highly imbalanced dataset like we have this will result in the majority class dominating in classification. - Below shows its confusion matrix, everything has been classified as not a hit. - This is further reflected by the model's F1 score of 0 - a metric combining the precision and recall of the model - this isis indicative that we are not fitting to the data at all.

`"balanced"` - Balanced class weighting will adjust the class weighting inversely proportional to the frequencies of each class. - This will assign a higher penalty to misclassifying a hit (class 1) and has resulted in a model that fits the dataset better. - This has still resulted in a rather low precision, with only 20% of classified hits being true hits.

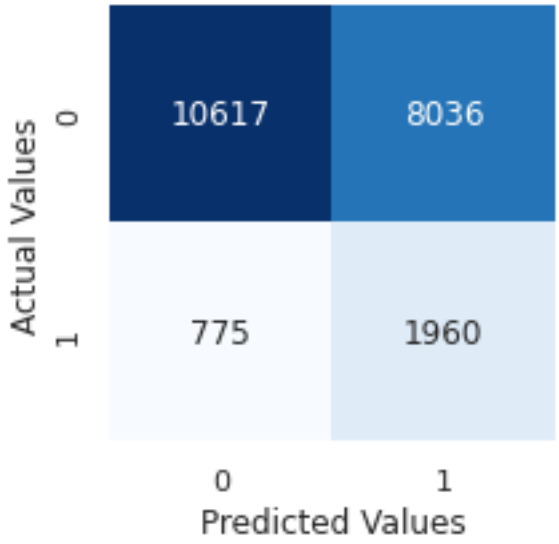
We ideally want to maximise the precision of song classification as hits. Due to the high cost in promoting and marketing songs, we only want to make this investment if a song truly has the potential to be a hit.

[8]:
```
for class_weight in ["None", "balanced"]:
  model = LogisticRegression(solver="newton-cg", class_weight=class_weight,
                             max_iter=1000)
  model.fit(X_train, y_train)
  y_pred = model.predict(X_test)

  print(f"CLASS WEIGHT: {class_weight}")
  print(f"F1 SCORE: {f1_score(y_test, y_pred)}")
  print(f"PRECISION: {precision_score(y_test, y_pred, zero_division=0)}\n")
  conf_matrix(y_test, y_pred)
```

```
CLASS WEIGHT: None
F1 SCORE: 0.0007309941520467837
PRECISION: 1.0
```




```
CLASS WEIGHT: balanced
F1 SCORE: 0.30790982640798054
PRECISION: 0.19607843137254902
```



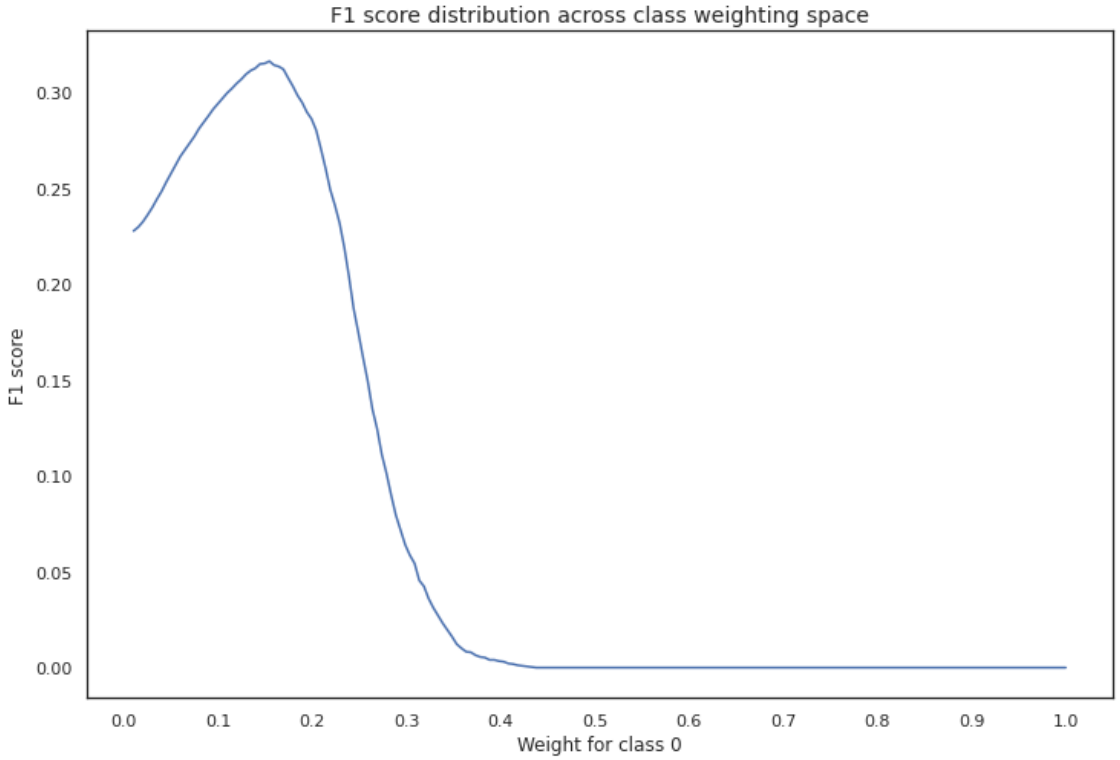
3.3 Hyperparameter Tuning

The `balanced` option seems like a reasonable heuristic for accounting for class imbalanced, although I have also performed a 5-fold stratified grid search over the class weighting parameter space from 0 - 1, with the goal of maximising the resultant model's F1 score.

[9]:
```
from sklearn.model_selection import GridSearchCV, StratifiedKFold

model = LogisticRegression(solver='newton-cg', penalty="l2")
weights = np.linspace(0,0.99,200)
weights_grid = [{1:x, 0:1.0-x} for x in weights]
param_grid = {'class_weight': weights_grid}

gridsearch = GridSearchCV(estimator=model,
                          param_grid=param_grid,
                          cv=StratifiedKFold(),
                          n_jobs=-1,
                          scoring='f1',
                          verbose=2).fit(X_train, y_train)

print(f"\nBEST MODEL PARAMETERS: {gridsearch.best_params_}")
print(f"MAX F1 SCORE: {gridsearch.best_score_}")
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed:   10.2s
[Parallel(n_jobs=-1)]: Done 158 tasks      | elapsed:   43.0s
[Parallel(n_jobs=-1)]: Done 361 tasks      | elapsed:  1.7min
[Parallel(n_jobs=-1)]: Done 644 tasks      | elapsed:  3.2min
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed:  4.6min finished


BEST MODEL PARAMETERS: {'class_weight': {1: 0.8457286432160804, 0:
0.15427135678391957}}
MAX F1 SCORE: 0.31635103473851
```

[13]:
```
weight_data = pd.DataFrame({'score': gridsearch.cv_results_['mean_test_score'],
                            'weight': (1- weights)})

plt.figure(figsize=(12,8))
sns.set_style('white')
sns.lineplot(x='weight', y='score', data=weight_data)
plt.xlabel('Weight for class 0')
plt.ylabel('F1 score')
plt.xticks([round(i/10,1) for i in range(0,11,1)])
plt.title('F1 score distribution across class weighting space', fontsize=14)
plt.show()
```



[11]:
```
print(gridsearch.best_estimator_)
```

```
LogisticRegression(C=1.0,
                   class_weight={0: 0.15427135678391957, 1: 0.8457286432160804},
                   dual=False, fit_intercept=True, intercept_scaling=1,
                   l1_ratio=None, max_iter=100, multi_class='auto', n_jobs=None,
                   penalty='l2', random_state=None, solver='newton-cg',
                   tol=0.0001, verbose=0, warm_start=False)
```

3.4 Final Model

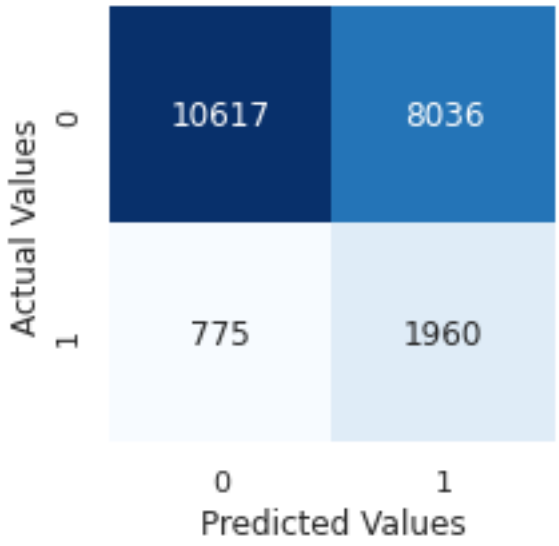
The resultant model from this optimisation is presented below, while accounting for class weighting has improved the F1 score of our model, it is still misclassifying a large portion of our dataset.

This is indicative of two points, either: - our model is too simplistic to fit our data; or - the features in our model are of low predictive power.

I believe that currently, the model has too few features to model popularity accurately. For example, an artist name has an impact on a song's number of streams. A mainstream artist will get more streams than an identical release from an indie artist, discounting promotion and marketing differences, simply from their following/fanbase size. Before adjusting my data downloading pipeline, I have built a series of more complex models to confirm this hypothesis.

[12]:
```
max_f1_score = gridsearch.best_score_
opt_weight = weight_data.loc[weight_data['score'] == max_f1_score,
                             'weight'].iloc[0]
optimum_weights = {0: opt_weight, 1: 1-opt_weight}

model = LogisticRegression(solver="newton-cg", class_weight = optimum_weights,
                           max_iter=1000)
model.fit(X_train, y_train)
predictions = model.predict(X_test)

print(f"CLASS WEIGHT: {optimum_weights}")
print(f"F1 SCORE: {f1_score(y_test, y_pred)}")
print(f"PRECISION: {precision_score(y_test, y_pred, zero_division=0)}")

conf_matrix(y_test, y_pred)

print(classification_report(y_test, predictions))
```

```
CLASS WEIGHT: {0: 0.15427135678391957, 1: 0.8457286432160804}
F1 SCORE: 0.30790982640798054
PRECISION: 0.19607843137254902
```




```
              precision    recall  f1-score   support

         0.0       0.92      0.69      0.79     18653
         1.0       0.22      0.58      0.31      2735

    accuracy                           0.68     21388
   macro avg       0.57      0.64      0.55     21388
```

```
weighted avg       0.83      0.68      0.73     21388
```

4 Random Forest Classifier

Before determining whether more data / new features are required to improve the predictive power of the models, I have built a random forest classifier. This utilises the concept of bootstrapping by repeatedly sampling distributions to build a more complex model than what is possible with standard logistic regression.

[1]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
from google.colab import drive
%pip install vaderSentiment --quiet
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, precision_score, recall_score
from sklearn.metrics import f1_score, classification_report
from sklearn.model_selection import RandomizedSearchCV

from scipy.stats import randint as sp_randint
drive.mount('/content/gdrive', force_remount=True)
%cd /content/gdrive/MyDrive/spotify/scripts
```

```
     |                | 133kB 6.6MB/s
Mounted at /content/gdrive
/content/gdrive/MyDrive/spotify/scripts
```

4.1 Feature Engineering

As in the prior logistic regression model, I have created some simple features from each audio feature in the dataset and removed any columns that are strings. Each column has then been normalised to speed up algorithm convergence. I have once again used a stratified K-fold, which forms a cross-validation over 5 folds to tune hyperparameters of my model.

[2]:
```
df = pd.read_csv("../data/master_data.csv")

audio_features = ['acousticness', 'danceability', 'energy', 'instrumentalness',
       'liveness', 'loudness', 'speechiness', 'tempo', 'valence', 'duration_ms']

df['year'] = df['release_date'].apply(lambda x: int(x[:4]))
df['popularity'] = df['popularity'].apply(lambda x: 1 if x>=50 else 0)

# Sentiment analysis of track title using VADER
```

```
analyser = SentimentIntensityAnalyzer()
df['sentiment'] = df['name'].apply(lambda x: analyser.
 ↪polarity_scores(x)['compound'])

duration_mean = df['duration_ms'].mean()
df['duration_ms'].fillna(duration_mean, inplace=True)

print(pd.isnull(df).sum())
df.drop(labels=["track_number", "artist_name", "album", "name", "id", "uri",
                "release_date"], axis=1, inplace=True)

from sklearn.preprocessing import MinMaxScaler
x = df.values
min_max_scaler = MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(x)
df = pd.DataFrame(x_scaled, columns = [*audio_features, "popularity", "year",
                                       "sentiment"])

neg, pos = np.bincount(df['popularity'])
total = pos + neg
# Assign class weights inversely proportional to count frequency of each class.
weight_for_0 = (1 / neg) * (total / 2.0)
weight_for_1 = (1 / pos) * (total / 2.0)
class_weight = {0: weight_for_0, 1: weight_for_1}

print('WEIGHT FOR CLASS 0: {:.2f}'.format(weight_for_0))
print('WEIGHT FOR CLASS 1: {:.2f}'.format(weight_for_1))
```

```
artist_name         0
album               0
track_number        0
id                  0
name                0
uri                 0
acousticness        0
danceability        0
energy              0
instrumentalness    0
liveness            0
loudness            0
speechiness         0
tempo               0
valence             0
duration_ms         0
release_date        0
popularity          0
year                0
sentiment           0
```

```
dtype: int64
WEIGHT FOR CLASS 0: 0.57
WEIGHT FOR CLASS 1: 3.91
```

[3]:
```
x = df.drop(["popularity"], axis=1)
print('FEATURES:', x.columns)
y = df.loc[:,"popularity"].values
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

print(f"\nTraining features shape: {X_train.shape}.")
print(f"Training labels shape: {y_train.shape}.")
print(f"Test features shape: {X_test.shape}.")
print(f"Test labels shape: {y_test.shape}.")
```

```
FEATURES: Index(['acousticness', 'danceability', 'energy', 'instrumentalness',
       'liveness', 'loudness', 'speechiness', 'tempo', 'valence',
       'duration_ms', 'year', 'sentiment'],
      dtype='object')

Training features shape: (57033, 12).
Training labels shape: (57033,).
Test features shape: (14259, 12).
Test labels shape: (14259,).
```

[4]:
```
def conf_matrix(y_test, pred_test):
    con_mat = confusion_matrix(y_test, pred_test)
    con_mat = pd.DataFrame(con_mat, range(2), range(2))

    plt.figure(figsize=(3,3))
    sns.set(font_scale=1)
    sns.heatmap(con_mat, annot=True, annot_kws={"size": 12}, fmt='g',
                cmap='Blues', cbar=False)
    plt.ylabel("Actual Values")
    plt.xlabel("Predicted Values")
    plt.show()
```

4.2 Random Forest Model

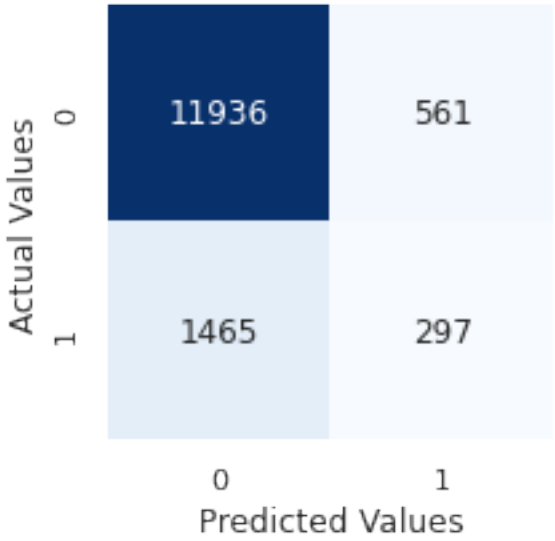
Let's begin by creating a baseline model to compare against our hyperparameter-tuned model. Even using the default settings for the RF model, we observe over a 10% increase in precision compared to the prior logistic regression model.

I have chosen precision as my metric of interest in this case, as we want to maximise the proportion of records that are true hits across all songs that are labelled as hits. This is because of the high cost of marketing required to promote a song, we want to make sure that a song truly has the potential to be successful before staking this upfront cost..

[7]:
```
baseline_rf = RandomForestClassifier(class_weight="balanced_subsample")
baseline_rf.fit(X_train, y_train)
```

```
y_pred = baseline_rf.predict(X_test)
print(baseline_rf.get_params())
print(f"F1 SCORE: {f1_score(y_test, y_pred)}")
print(f"PRECISION: {precision_score(y_test, y_pred, zero_division=0)}\n")
conf_matrix(y_test, y_pred)
```

```
{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': 'balanced_subsample',
'criterion': 'gini', 'max_depth': None, 'max_features': 'auto',
'max_leaf_nodes': None, 'max_samples': None, 'min_impurity_decrease': 0.0,
'min_impurity_split': None, 'min_samples_leaf': 1, 'min_samples_split': 2,
'min_weight_fraction_leaf': 0.0, 'n_estimators': 100, 'n_jobs': None,
'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
F1 SCORE: 0.2267175572519084
PRECISION: 0.34615384615384615
```



As in the logistic regression model prior, I have used a similar random search strategy to tune my model. Instead of focusing on class_weight in this case, I have looked at optimising the number of trees, their depth and the minimum number of samples per split.

[ ]:
```
rf = RandomForestClassifier(random_state=42, n_jobs=-1,␣
 ↪class_weight='balanced_subsample')
param_dist = {"max_depth": [3, None],
              "max_features": sp_randint(1, X_train.shape[1]),
              "min_samples_split": sp_randint(2, 11),
              "bootstrap": [True],
              "n_estimators": sp_randint(250, 1000)}

random_search = RandomizedSearchCV(rf, param_distributions=param_dist,
```

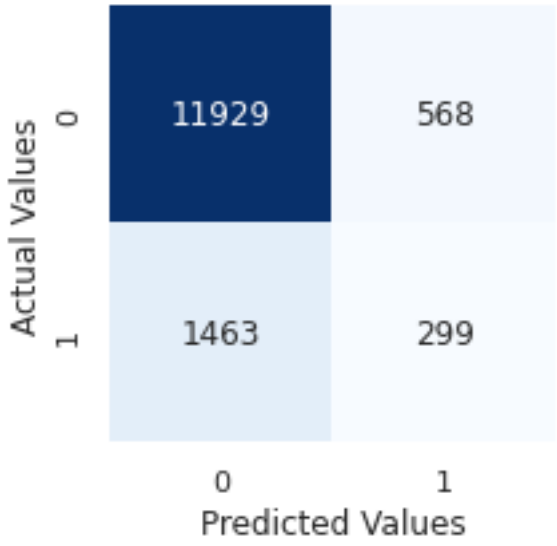
```
                                    n_iter=20, cv=5, iid=False, verbose=1,
                                    random_state=42).fit(X_train, y_train)
print(random_search.best_params_)
```

[ ]:
```
print(random_search.best_params_)
```

```
{'bootstrap': True, 'max_depth': None, 'max_features': 5, 'min_samples_split':
2, 'n_estimators': 709}
```

[11]:
```
#best_rf = random_search.best_estimator_
best_rf = RandomForestClassifier(bootstrap=True, max_depth=None, max_features=5,
          min_samples_split=2, n_estimators=709, random_state=42, n_jobs=-1,
          class_weight='balanced_subsample')
best_rf.fit(X_train, y_train)
y_pred = best_rf.predict(X_test)
print(f"F1 SCORE: {f1_score(y_test, y_pred)}")
print(f"PRECISION: {precision_score(y_test, y_pred, zero_division=0)}")
print(f"RECALL: {recall_score(y_test, y_pred, zero_division=0)}")
conf_matrix(y_test, y_pred)
```

```
F1 SCORE: 0.2274629136553823
PRECISION: 0.34486735870818913
RECALL: 0.16969353007945517
```



4.3 Final Conclusions

While it may appear that this model performs worse than the prior logistic regression, it has a significantly lower number of false positives, which I believe is far more important than the model's overall F1 Score, for the aforementioned costs of promoting a song that will not be a success regardless of the time or marketing placed behind it.

The model however still does not fit the dataset well [it is still only identifying ~20% of the true positives (hits)], the use of a random forest allows for significantly more complex behaviours to be learned by extending the number and depth of the tress in the ensemble. I have also built a series of neural networks in Keras that have not presented in this notebook, as they were not fitting to the dataset with sufficient precision to be worthy of presentation. It is clear that the current features in our model (which are specific to the song's structure and style) are not sufficiently powerful to predict whether a song will be a hit or not.

Given more time I would have adjusted my data collection pipeline to pull information about the artist for each song (such as their number of followers), while also using Billboard's API to identify whether a specific song is featured in their Top 100 or not, as this is a slightly fairer allocation of whether a song is a success or not. This set of notebooks has further highlighted to me the complexities of whether or not a track is a success. This is clearly due to the nuances of each genre and the specific target demographic that will determine a song's success.