Capstone 1

Building a Song Preference Prediction Model

By: Aaron Kochman

Problem Statement

Recommender systems are developed using a variety of machine learning techniques with varying accuracies. Building a robust recommender system that is personalized is imperative to keep users interacting with the specific application the recommender system is acting upon.

Concept of Capstone Project

The goal of this capstone project is to develop a model that will predict if a Spotify user will like or dislike a song. Essentially this is the rudimentary step in developing a song recommendation playlist similar to Spotify's "Discover Weekly Playlist". In order to develop the prediction model, data needed to be available. Conceptually it was determined that songs could be sorted into two playlists based on user preference, a "Like" and "Dislike" playlist. Once each playlist had a sufficient amount of tracks, music data accessible via the Spotify Web API, Spotipy, could be acquired and analyzed to develop an accurate prediction model.

A recommender system provides personalized recommendations of content to users. Recommender systems are a powerful tool for content driven businesses as well as retailers. Music streaming providers require these systems to suggest music that appeal to individuals based on previous music they have consumed. The recommender systems are driven by a variety of machine learning algorithms that vary in complexity and accuracy. Using machine learning techniques to recommend likable song selections is an important portion to commercial music streaming platforms like Spotify, Apple Music, and Pandora among others. The driving motivation behind accurate recommendation algorithms is to keep the consumer on the desired platform.

Spotify has achieved success with their recommender system "Discover Weekly". If Spotify is recommending great music to a user via the Discover Weekly playlist, the listener will then be more inclined to keep listening on their streaming service. In this Capstone I plan on familiarizing myself with techniques in data wrangling, data analysis, data visualization, and machine learning that will help to develop a powerful and capable recommender system like Spotify's Discover Weekly playlist.

Business Case

An accurate recommender system can be extremely useful to a variety of businesses that rely on consumer preference to profit. In the case of Netflix, YouTube, or Spotify, recommended content can subsequently draw user's to the platform for longer periods of time. Metrics can be derived from the recommended content that can produce business use cases for different recommender systems. In the case of YouTube, the recommender system can be analyzed by the number of clicks or interactions that users have with recommended content. More interactions would indicate a more accurate or better recommender system that would lead to more users spending more time, which would equate to more advertising revenue for the company.

Summary of Capstone Project

A playlist of songs that I like and a playlist of songs that I dislike were created to develop this model based on my personal preference in music and was completed in my Spotify account. Spotify's Web API Spotipy was used to obtain track analytics for each song within the two playlists. The desired information was stored in separate Pandas data frames during the data wrangling phase and a "user_preference" column was added with "1" being associated with songs that I liked and "0" being associated with songs that I disliked. Once the desired music data was pulled from Spotipy, the data frames were merged together for analysis. Music data including danceability, energy, loudness, speechiness, acousticness, valence, instrumentalness, and tempo were all analyzed respectively to determine which would be suitable to use for a song preference prediction model. After analysis and determining that each playlist was vastly different in all fields, I decided to use song popularity, danceability, energy, and loudness as the fields that each classifier would base it's accuracy for song preference on. The classifiers used in this project were logistic regression, random forests, decision trees, support vector machines, and k-nearest neighbors (k-NN). Overall all of the classifiers yielded accuracy above 90%.

A new playlist was set up that contained potential songs that I would either like or dislike based on Spotify's existing Discover Weekly Playlist that it had set up for me. Assuming that I would like most or all of these songs, I added some songs that I did not prefer, and ran a logistic regression prediction model on the list. Once the logistic regression prediction model classified each song in a binary format (like = 1, dislike = 0) After listening to a sample of the prediction models classified tracks, I concluded that it was very accurate in determining if I like a song or dislike a song. This same method could be utilized by any individual for any playlist.

Data Wrangling

Spotify Account and Playlist Creation

Spotify tracks, playlists, usernames, and every component of their database contains a unique ID that can be queried by applications using the Spotify API or Spotipy. For this project I used my username (username = ‘ernflerberg’) as well as playlist ID’s that I created for songs that I like and songs that I dislike. Songs that I liked were placed into playlist “Liked” ("3s3OCt230DDEIGX8xOY58A") while songs that I disliked were placed into playlist “Dislike” with ID ("0VDhUjxtx7ZzErPwrJcLCH").

Connecting to the Spotipy API using an Authorization Code Flow

Based on the Spotipy API documentation, connecting to the Spotify Web API requires the following:

1. **USERNAME**: Spotify account id.
2. **CLIENT_ID**: The client id from the Spotify for Developers page.
3. **CLIENT_SECRET**: The secret generated for the specific Spotify Application on your Spotify Developer page.

The credentials were kept in a separate config file named “config_ernflerberg.cfg”. This allowed for no sign in to occur and results to be pulled using the library “configparser”. This was performed mainly because suggested methods for setting up the credentials were unreliable and failed upon re-running the code.

```
In [2]: #Authenticate with config file and spotipy client_id, client_secret, username, values

config = configparser.ConfigParser()
config.read('config_ernflerberg.cfg')
client_id = config.get('SPOTIFY', 'CLIENT_ID')
client_secret = config.get('SPOTIFY', 'CLIENT_SECRET')
username = config.get('SPOTIFY', 'USERNAME')


auth = oauth2.SpotifyClientCredentials(
    client_id=client_id,
    client_secret=client_secret
)

token = auth.get_access_token()
spotify = spotipy.Spotify(auth=token)
```

Figure 1: configparser set to username = ‘ernflerberg’

Data Collection for Each Playlist - JSON to Pandas data frame from Spotipy

According to the Spotipy API documentation, we needed to set up our data wrangling to work with the Spotify URI's and ID's. The unique playlist URI (Uniform Resource Identifier) was used to pull song information from the API in a JSON format as shown in Figure 2.

```
In [57]: playlistDictionary={
                         "Liked": "3s3OCt23ODDEIGX8xOY58A",  #like: spotify:user:ernflerberg:playlist:3s3O
         Ct23ODDEIGX8xOY58A
                         "Dislike": "0VDhUjxtx7ZzErPwrJcLCH" #dislike: spotify:user:ernflerberg:playlist:3
         tfIr2Q4Qq10fsTzk3lHt6
                         }
         #like: https://open.spotify.com/user/ernflerberg/playlist/3s3OCt23ODDEIGX8xOY58A?si=5n5ZRzWwQSCE8ULv
         fEbLiQ
         #dislike: https://open.spotify.com/playlist/4h4W1o7xGfgd60dOeME5aq?si=33-HHYLrTq69we4MYRb7WQ

In [12]: #Pull 'disLike' Playlist
         #spotify:playlist:3tfIr2Q4Qq10fsTzk3lHt6

         uri = 'spotify:user:ernflerberg:playlist:0VDhUjxtx7ZzErPwrJcLCH'
         username = username
         disliked_playlist_id = '0VDhUjxtx7ZzErPwrJcLCH'
         disliked_results = spotify.user_playlist(username, disliked_playlist_id)

In [13]: #Pull 'Like' Playlist

         uri = 'spotify:user:ernflerberg:playlist:3s3OCt23ODDEIGX8xOY58A'
         username = username
         liked_playlist_id = '3s3OCt23ODDEIGX8xOY58A'
         liked_results = spotify.user_playlist(username, liked_playlist_id)

In [14]: #Check that the 'Liked' playlist connects to a song
         liked_results['tracks']['items'][0]['track']['id']

Out[14]: '6yl8Es1tCYD9WdSkeVLFw4'

In [15]: #Check that the 'Disliked' playlist connects to a song
         disliked_results['tracks']['items'][0]['track']['id']

Out[15]: '0cm7kloidR7elOAsWnkLZE'
```

Figure 2: Overall structure to pull data from tracks in a playlist.

Once we were able to identify individual playlists, we could query track id's to understand that our request worked. We utilized this method to pull track information in a JSON format, appending to a pandas dataframe as shown in Figure 3. Note that a "user_preference" column was added as well for liked/disliked information. (1=liked, 0=disliked)

In [16]:

```
#Pull the 'Liked' playlist trac  k information using the Spotipy API JSON
ldf = []
for i in liked_results['tracks']['items']:
    ldf.append([i['track']['id'],
                i['added_at'],
          i['track']['album']['artists'][0]['name'],
          i['track']['album']['name'],
          i['track']['duration_ms'],
          i['track']['name'],
          i['track']['popularity']])
ldf = pd.DataFrame(ldf)
#Add column names for the "Liked" Playlist
ldf.columns = ['song_id','added_at','artist','album','duration_ms','songname','popularity']
#Add column "user_preference" to the Liked Playlist
ldf['user_preference'] = 1
ldf.head()
```

Out[16]:

	song_id	added_at	artist	album	duration_ms	songname	popularity	user_pre
0	6yl8Es1tCYD9WdSkeVLFw4	2019-06-06T05:11:15Z	AC/DC	Who Made Who	210880	You Shook Me All Night Long	71	1
1	3fkPMWQ6cBNBLuFcPyMS8s	2019-06-06T05:11:15Z	Blue Öyster Cult	Fire of Unknown Origin	271000	Burnin' for You	64	1
2	5EWPGh7jbTNO2wakv8LjUI	2019-06-06T05:11:15Z	Lynyrd Skynyrd	Pronounced' Leh-'Nerd 'Skin-'Nerd	547106	Free Bird	69	1
3	6QDbGdbJ57Mtkflsg42WV5	2019-06-06T05:11:15Z	Van Halen	1984 (Remastered)	282746	Hot for Teacher - 2015 Remaster	66	1
4	6J17MkMmuzBilOjRH6MOBZ	2019-06-06T05:11:15Z	AC/DC	Back In Black	266040	Rock and Roll Ain't Noise Pollution	60	1

Figure 3: JSON data wrangling from specified playlist stored in a pandas data frame. The “user_preference” column is shown on the right.

Once the basic song information was pulled from the API, additional audio features were then requested from the API based on song_id, and appended to the same data frame. (Figure 4)

```
In [19]: #Pull Audio Features from Spotipy on song_id for Liked playlist
laf = []
for i in ldf.song_id:
    x = sp.audio_features(i)
    laf.append([i,
               x[0]['danceability'],
                x[0]['energy'],
                    x[0]['key'],
                    x[0]['loudness'],
                    x[0]['mode'],
                    x[0]['speechiness'],
                    x[0]['acousticness'],
                    x[0]['instrumentalness'],
                    x[0]['liveness'],
                    x[0]['valence'],
                    x[0]['tempo'],
                    x[0]['time_signature']])
laf = pd.DataFrame(laf)

laf.columns = ['song_id',
             'danceability',
             'energy',
              'key',
              'loudness',
              'mode',
              'speechiness',
              'acousticness',
              'instrumentalness',
              'liveness',
              'valence',
              'tempo',
              'time_signature'
             ]

#Merge the Liked Audio Features and track information together
lpaf = pd.merge(ldf, laf, on='song_id')
lpaf.head()
```

Out[19]:

song_id	added_at	artist	album	duration_ms	songname	popularity	u
6yI8Es1tCYD9WdSkeVLFw4	2019-06-06T05:11:15Z	AC/DC	Who Made Who	210880	You Shook Me All Night Long	71	1
3fkPMWQ6cBNBLuFcPyMS8s	2019-06-06T05:11:15Z	Blue Öyster Cult	Fire of Unknown Origin	271000	Burnin' for You	64	1
	2019-06-	Lynyrd	Pronounced'				

Figure 4: Additional audio features pulled from Spotipy via the song_id and appended to the existing data frames for the “liked” and “disliked” playlists.

The two playlist data frames were combined into a final data frame (df_combined) which could then be used for analysis in a clean format. (Figure 5)

```
In [21]: #Join Disliked Playlist to Liked Playlist in New Data Frame
df_combined = pd.concat([lpaf, dpaf])
df_combined.head()
```

Out[21]:

	song_id	added_at	artist	album	duration_ms	songname	popularity
0	6yl8Es1tCYD9WdSkeVLFw4	2019-06-06T05:11:15Z	AC/DC	Who Made Who	210880	You Shook Me All Night Long	71
1	3fkPMWQ6cBNBLuFcPyMS8s	2019-06-06T05:11:15Z	Blue Öyster Cult	Fire of Unknown Origin	271000	Burnin' for You	64
2	5EWPGh7jbTNO2wakv8LjUI	2019-06-06T05:11:15Z	Lynyrd Skynyrd	Pronounced' Leh-'Nerd 'Skin-'Nerd	547106	Free Bird	69
3	6QDbGdbJ57Mtkflsg42WV5	2019-06-06T05:11:15Z	Van Halen	1984 (Remastered)	282746	Hot for Teacher - 2015 Remaster	66
4	6J17MkMmuzBilOjRH6MOBZ	2019-06-06T05:11:15Z	AC/DC	Back In Black	266040	Rock and Roll Ain't Noise Pollution	60

Figure 5: The “Liked” and “Disliked” playlists combined into a singular data frame, “df_combined”

Data Story

Exploring the Merged Playlist (“df_combined”)

Once the “Liked” playlist and “Dislike” playlist were combined into a singular data frame with a “user_preference” column and song attributes, we could explore the data and visualize some of the significantly distinct attributes.

```
[68]: df_disliked = df_combined[df_combined['user_preference']==0]
      df_liked = df_combined[df_combined['user_preference']==1]


      sns.set(style='darkgrid', palette="deep", font_scale=1.1, rc={"figure.figsize": [8, 5]})
      sns.distplot(df_disliked['acousticness'], norm_hist=True, kde=True, bins=20, hist_kws={"alpha": 1}).set(xlabel='acousticness', ylabel='Count')
      sns.distplot(df_liked['acousticness'], norm_hist=True, kde=True, bins=20, hist_kws={"alpha": 1}).set(xlabel='acousticness', ylabel='Count')
      plt.legend(title='User Preference', labels=['Disliked', 'Liked'])
```

```
[68]: <matplotlib.legend.Legend at 0x1c027361f08>
```

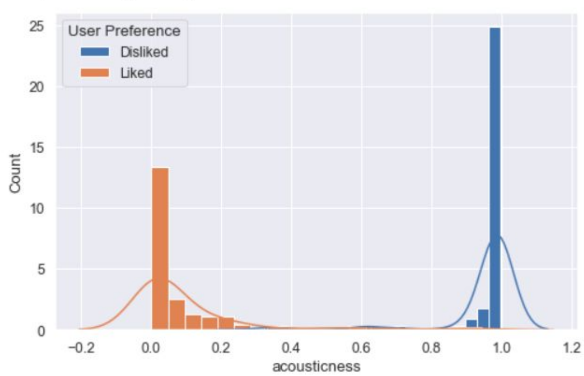


Figure 1: Song attribute “acousticness” shows a distinct difference between the two playlist preferences.

Using the “user_preference” column to classify the “Liked” and “Disliked” playlists respectively, we can see that the song attributes “acousticness”, “danceability”, “energy”, and “loudness” are distinct columns and are vastly different between each playlist. Initially there was too much overlap between each playlist regarding the song attributes, so more distinctive genres were used between the “Liked” and “Disliked” playlists to provide different song attribute results.

Song Feature Analysis

Danceability

The Liked playlist showed Danceability averaging around 0.5 while the Disliked playlist averaged 0.2.

Energy

The Liked playlist showed Energy averaging around 0.9 while the Disliked playlist averaged below 0.2.

Loudness

The Liked playlist showed Loudness mostly ranging from -10 to 0 db while the Disliked playlist remained in the -40 to -20 db range.

Spechiness

The Liked playlist showed Spechiness as a very similar feature that would need to remove some outliers to detect any minute differences.

Acousticness

The Liked playlist showed Acousticness averaging around 0.0 while the Disliked playlist averaged closer to 1.0.

Valence

The Liked playlist showed Valence averaging around 0.5 while the Disliked playlist averaged 0.2.

Instrumentalness

The Liked playlist showed Instrumentalness averaging around 0.0 while the Disliked playlist averaged 0.9.

Tempo

We can see that tempo fluctuated between each playlist but generally the songs in the Liked playlist had a higher tempo than songs in the Disliked playlist.

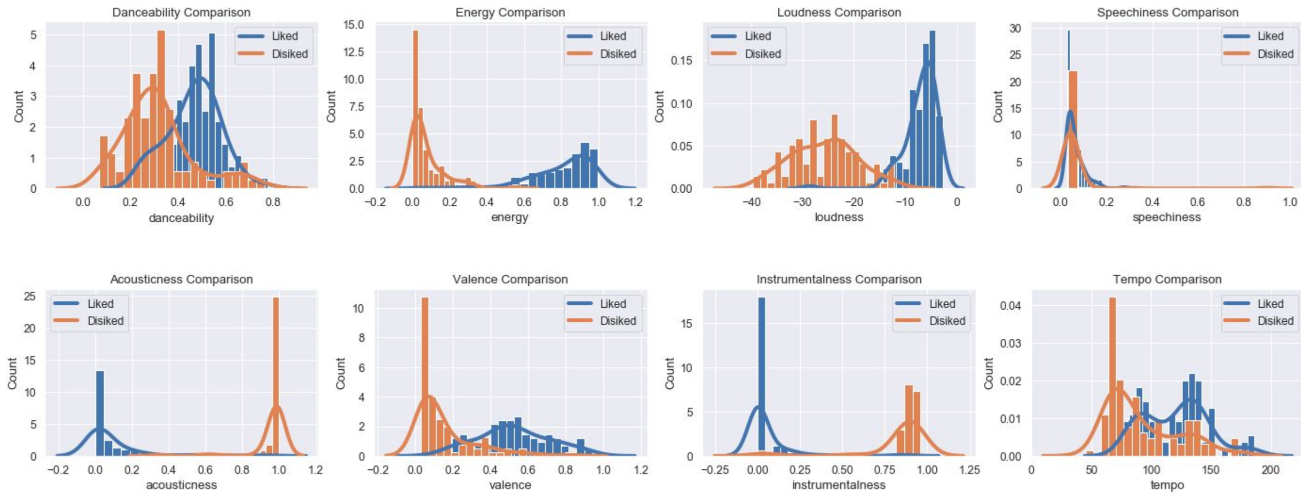


Figure 2: Each song attribute plotted using Seaborn's distribution plotting method.

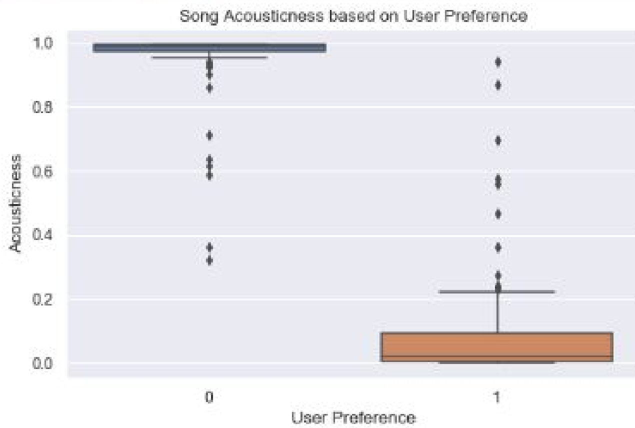
When each song feature is analyzed based on song preference, we can see that there are significant differences. In Figure 2 each song feature is shown in its own distribution plot based on user preference with the Liked songs in blue and the Disliked songs in orange.

```
In [46]: # plot a boxplot of acousticness for an overview
sns.boxplot(x='user_preference', y='acousticness', data=df_combined)

_ = plt.xlabel('User Preference')
_ = plt.ylabel('Acousticness')
_ = plt.title('Song Acousticness based on User Preference')

plt.show()

#add hypothesis test
```




```
In [47]: # plot a boxplot of tempo for an overview
sns.boxplot(x='user_preference', y='tempo', data=df_combined)

_ = plt.xlabel('User Preference')
_ = plt.ylabel('Tempo')
_ = plt.title('Song Tempo based on User Preference')

plt.show()
```

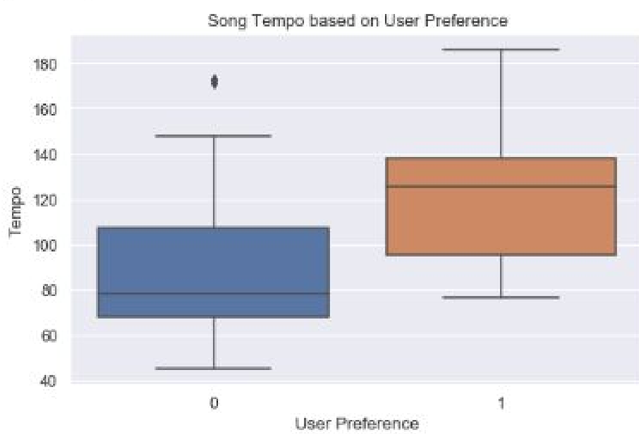


Figure 3: A box plot representing tempo based on the user preference. This user tends to like more songs with a higher tempo, while disliking songs with a lower tempo.

In general we can see that the “Liked” playlist consists of high energy, loud songs with low acousticness while the opposite is true for the “Disliked” playlist. This was done based on song genre’s to determine if a machine learning model could predict if you would like or dislike a song based on some of these features.

```
In [51]: # Pre-format DataFrame
stats_df = df_combined.drop(['song_id', 'added_at', 'artist', 'album', 'duration_ms', 'songname', 'popular
ity', 'tempo', 'key', 'mode','time_signature'], axis=1)
```

```
In [52]: def normalize(dataset):
    dataNorm=((dataset-dataset.min())/(dataset.max()-dataset.min()))*100
    dataNorm["user_preference"]=dataset["user_preference"]
    return dataNorm
```

```
In [53]: normdata=normalize(stats_df)
normdata.sample(5)
```

Out[53]:

	user_preference	danceability	energy	loudness	speechiness	acousticness	instrumentalness	liveness	valenc
34	0	16.126304	6.469464	39.223997	1.610692	97.891559	89.166667	5.013303	0.7608
13	0	25.993798	12.365429	52.679057	1.210875	98.694775	96.250000	10.096625	24.137
26	0	34.451649	1.116813	5.667862	1.325109	98.995981	81.875000	4.887271	16.418
19	1	75.190302	86.316531	93.267036	0.000000	0.035722	0.129167	7.029828	45.749
69	1	58.979419	90.542308	90.520052	1.656386	1.405305	43.854167	3.977034	63.281

```
In [54]: # Scatter plot
sns.lmplot(x='energy', y='loudness', data=normdata,
           fit_reg=False, # No regression line
           hue='user_preference')
```

Out[54]: <seaborn.axisgrid.FacetGrid at 0x5de88f0>

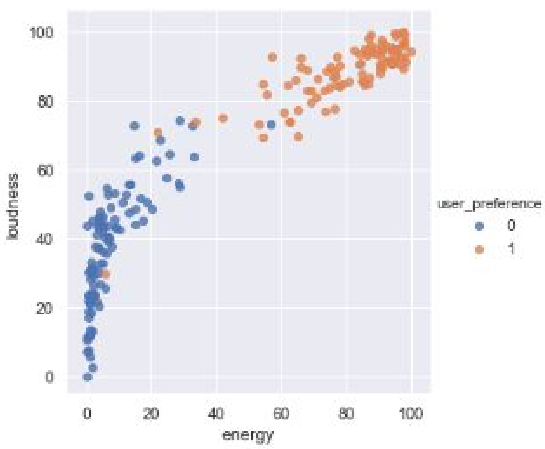


Figure 4: Normalized scatter plot based on user preference using the energy and loudness columns.

We can conclude from Figure 4 that the user tends to like loud music and dislike quieter music.

```
In [55]: sns.swarmplot(x='user_preference', y='energy', data=normdata)

Out[55]: <matplotlib.axes._subplots.AxesSubplot at 0x5e3fe50>
```

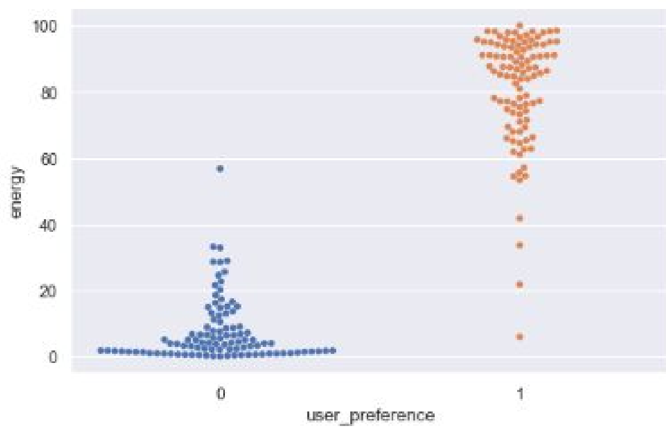


Figure 5: Swarm plot showing energy based on user preference.

We can see from the scatter plot in Figure 4 that there is a very clear correlation between similar attributes like energy and loudness that differentiate based on the user's preference of songs. Since this is what we are expecting, we can use these two very different playlists to produce an accurate machine learning prediction model.

Machine Learning

Setting up Training Data

```
# Import necessary modules
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.metrics import accuracy_score
```

```
#Set up Training Data
trainingData = df_combined[['popularity', 'danceability','energy','loudness', 'user_preference']]
trainingData.columns = ['popularity', 'danceability','energy','loudness', 'user_preference']
X = trainingData[['popularity', 'danceability','energy','loudness']]
y = trainingData['user_preference']
print(trainingData.shape)


train, test = train_test_split(trainingData, test_size = 0.30)
print("Training size: {}, Test size: {}".format(len(train),len(test)))
```

```
(200, 5)
Training size: 140, Test size: 60
```

In order to build a machine learning predictor suitable to predict if a user would like or dislike a future song, a variety of machine learning classification algorithms were analyzed. The classification models that were analyzed were:

- Logistic Regression
- Random Forests
- Decision Trees
- Support Vector Machines
- k-nearest neighbors (KNN)

Logistic Regression

```
#Logistic Regression
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=42)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)

# Create the classifier: logreg
logreg = LogisticRegression()

# Fit the classifier to the training data
logreg.fit(X_train, y_train)

# Predict the labels of the test set: y_pred
y_pred = logreg.predict(X_test)

# Compute and print the confusion matrix and classification report
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(accuracy_score(y_test, y_pred))
```

```
(140, 4) (60, 4) (140,) (60,)
[[29  0]
 [ 1 30]]
              precision    recall  f1-score   support

           0       0.97      1.00      0.98        29
           1       1.00      0.97      0.98        31

    accuracy                           0.98        60
   macro avg       0.98      0.98      0.98        60
weighted avg       0.98      0.98      0.98        60

0.9833333333333333
```

Random Forests

```
#Random Forests

# Import train_test_split function
from sklearn.model_selection import train_test_split

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3) # 70% training and 30% test

#Import Random Forest Model
from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier
clf=RandomForestClassifier(n_estimators=100)

#Train the model using the training sets y_pred=clf.predict(X_test)
clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)

#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9666666666666667
```

Decision Trees

```
#Decision Tree

from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
from sklearn.model_selection import train_test_split # Import train_test_split function
from sklearn import metrics #Import scikit-learn metrics module for accuracy calculation

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1) # 70% training and 30% test

# Create Decision Tree classifer object
clf = DecisionTreeClassifier()

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)


# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9833333333333333
```

Support Vector Machines

```
#Support Vector Machines

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,random_state=109) # 70% training and 30% test

#Import svm model
from sklearn import svm

#Create a svm Classifier
clf = svm.SVC(kernel='linear') # Linear Kernel

#Train the model using the training sets
clf.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)


#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics

# Model Accuracy: how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 1.0
```

k-nearest neighbors (KNN)

```
#KNN k=5

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3) # 70% training and 30% test

#Import knearest neighbors Classifier model
from sklearn.neighbors import KNeighborsClassifier

#Create KNN Classifier
knn = KNeighborsClassifier(n_neighbors=5)

#Train the model using the training sets
knn.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = knn.predict(X_test)

#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 1.0
```

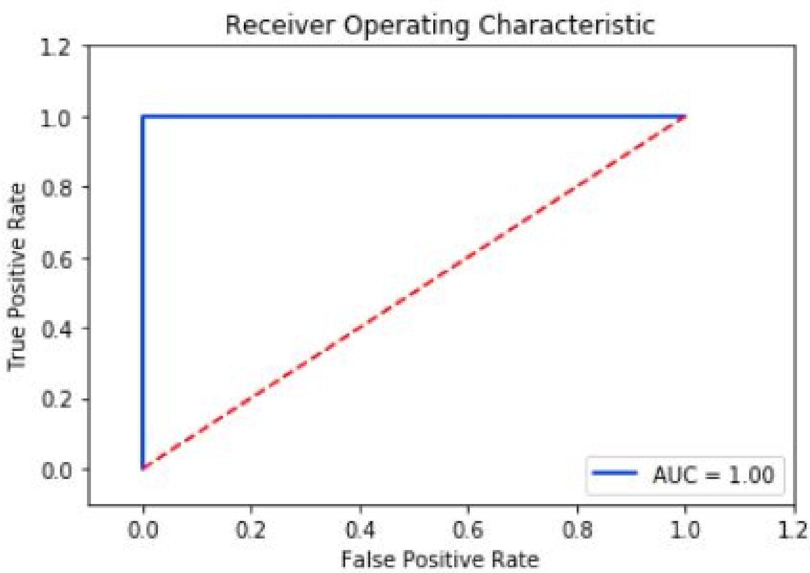
ROC AUC Score

```
#roc_auc_curve

from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import random

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)

plt.title('Receiver Operating Characteristic')
plt.plot(false_positive_rate, true_positive_rate, 'b',
label='AUC = %0.2f'% roc_auc)
plt.legend(loc='lower right')
plt.plot([0,1],[0,1],'r--')
plt.xlim([-0.1,1.2])
plt.ylim([-0.1,1.2])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



Prediction

```
trainingdatapredict = predictfeatures[['popularity', 'danceability','energy','loudness']]
```

```
predictarray = logreg.predict(trainingdatapredict)
predictarray
```

```
array([1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 0, 0, 1, 1, 1, 1, 1], dtype=int64)
```

```
likedSongs = 0
i = 0
for prediction in predictarray:
    if(prediction == 1):
        print ("Song: " + predictfeatures["songname"][i] + ", By: "+ predictfeatures["artist"][i])
        #sp.user_playlist_add_tracks("1287242681", "7eIX1zvtpZR3M3rYFVA7DF", [test['id'][i]])
        likedSongs= likedSongs + 1
    i = i +1
```


```
likedSongs = 0
i = 0
for prediction in predictarray:
    if(prediction == 1):
        print ("Song: " + predictfeatures["songname"][i] + ", By: "+ predictfeatures["artist"][i])
        #sp.user_playlist_add_tracks("1287242681", "7eIX1zvtpZR3M3rYFVA7DF", [test['id'][i]])
        likedSongs= likedSongs + 1
    i = i +1
```


```
likedSongs
```

```
Song: Cringe, By: Matt Maeson
Song: Milk and Honey, By: LUTHI
Song: Cooler, By: Mikey Mike
Song: Clean (feat. The Grouch), By: J.Lately
Song: Heat of the Summer, By: Young the Giant
Song: 4th of July, By: Cold War Kids
Song: Fearless, By: SonReal
Song: Hi Power, By: The Late Ones
Song: Way Down, By: The Dangerous Summer
```

```
dislikedSongs = 0
i = 0
ResultList = []
for prediction in predictarray:
    if(prediction == 0):
        print ("Song: " + predictfeatures["songname"][i] + ", By: "+ predictfeatures["artist"][i])
        #sp.user_playlist_add_tracks("1287242681", "7eIX1zvtpZR3M3rYFVA7DF", [test['id'][i]])
        dislikedSongs= dislikedSongs + 1
    i = i +1

dislikedSongs
```

```
Song: Feel Like A Stranger > - Live 2018-03-16, By: Joe Russo's Almost Dead
Song: All Around the World, By: Mickey Avalon
Song: Prayer for the Day, By: TreeHouse!
Song: How Sweet It Is (To Be Loved By You) - Live at French's Camp, Piercy, CA, 8/29/1987, By: Jerry Garcia Band
Song: Flamenco Sketches, By: Miles Davis
Song: That Old Feeling, By: Chet Baker
Song: A Nightingale Sang In Berkeley Square, By: Stan Getz
Song: Stranger to Me, By: Paul Luc
Song: Koo Koo, By: Toots & The Maytals
Song: Headcase - Acoustic Version, By: Abhi The Nomad
Song: Hitch-Hiker's Hero, By: Atlanta Rhythm Section
Song: I Loved Being My Mother's Son, By: Purple Mountains
Song: Ashes to Ashes, By: Jenny Hval

13
```

During the prediction phase, each prediction model produced accuracy results over 90%. The Logistic Regression model was chosen because of the high accuracy (98%) it produced while still having 2% chance of error.

When trained with the “Liked” and “Disliked” playlists and applied to the prediction playlist of potential songs, the Logistic Regression Prediction Model predicted that out of 98 songs, I would like 85 songs (87%) and dislike 13 songs (13%). This is consistent with the fact that most of my prediction playlist is made up of songs that Spotify’s Discover Weekly Playlist had recommended to me previously.

Conclusion

After listening to the new playlist of liked songs produced by the Logistic Regression model, I would agree that most of the songs I would realistically add to a “Liked” playlist however there were songs that I did not like. After some thought I concluded that the Logistic Regression model could not have predicted that I was not necessarily a Country genre fan, even though the song features for a Country song may be very similar to a Rock song. This added layer of complexity would need to be quantified and applied to the model if a new and improved model was to be developed.

This sort of unforeseen or unaccounted for metric could lead to false positives in the analytics of the model. The model may result in 98% accuracy when in reality the actual accuracy is closer to 70%. User preference is subjective and song features or other metrics could dictate that a user would like a song, when in reality they may not like the pitch of the singer's voice. Another possibility for a false positive would be if a person does not like an artist or lyrics, but would like danceability or tempo of the song. These factors could have data collection methods and be applied to a predictive model, but would require more data, more time, and more resources.

Industries profit from predictive models because it keeps users engaged with their application or platform. In the case of Spotify, they are not only taking into account the data driven makeup of each song, but are also determining listening patterns, tracking likes/dislikes from users, recommending songs that other users have liked with similar listening histories, as well as a variety of other techniques that can be applied to their predictive models.