In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
import scipy.stats as stats
from sklearn.decomposition import PCA as sk_PCA
from sklearn import svm

import sqlite3
import matplotlib.pyplot as plt
from PIL import Image
%matplotlib inline
import seaborn as sns
import requests
from sklearn import datasets
from sklearn.decomposition import PCA as sk_PCA
from sklearn.cluster import KMeans as sk_KMeans, DBSCAN as sk_DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import pairwise_distances_argmin
from sklearn.metrics import silhouette_samples, silhouette_score
import scipy.stats as stats
from sklearn.model_selection import train_test_split

from sklearn.pipeline import Pipeline
sns.set()
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
#random seed
#CHANGE TO N NUMBER
np.random.seed(13839901)
```

In [3]:
```
df = pd.read_csv('spotify52kData.csv', index_col = 'songNumber')
```

Question 1 - Is there a relationship between song length and popularity of a song?

In [4]:
```
df_duration = df[['artists', 'track_name', 'popularity','duration']]
```

In [5]:
```
df_duration = df_duration.drop_duplicates()
```

In [6]:
```
df_duration.shape
```

Out[6]:
```
(44082, 4)
```

In [7]:
```
#here we see there is no NaN in popularity
np.any(np.isnan(df_duration['popularity']))
```

Out[7]:
```
False
```

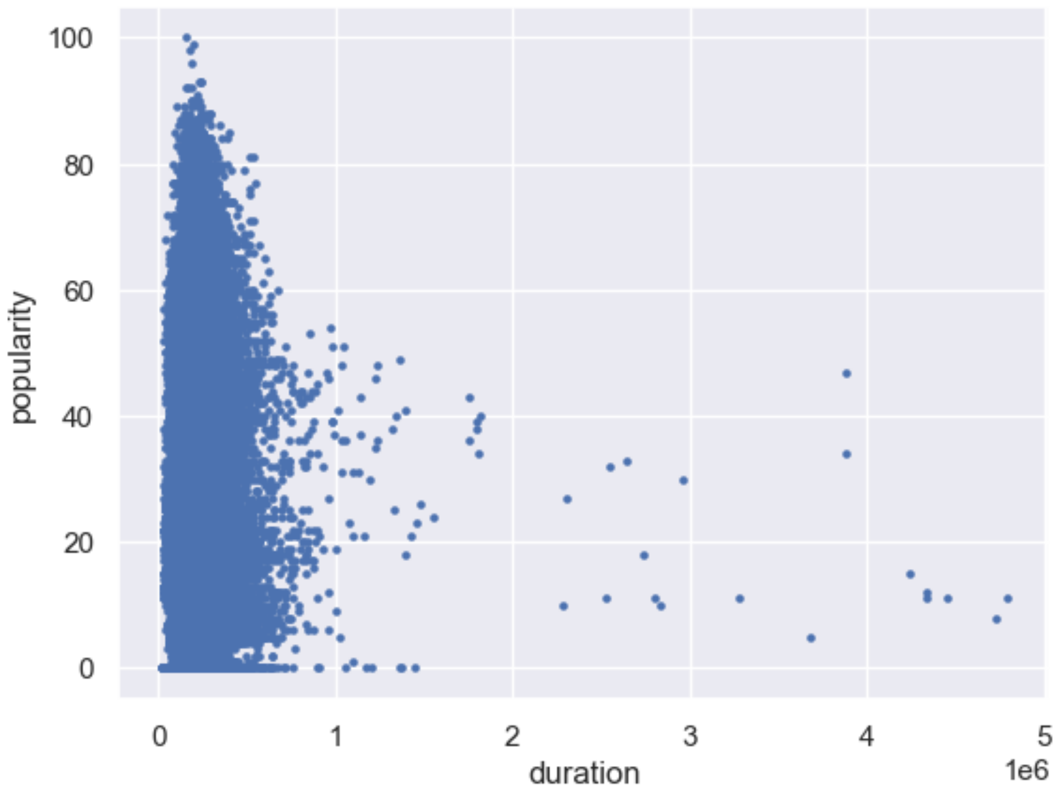
In [8]:
```
popularity = np.array(df_duration['popularity'])
```

In [9]:
```
duration = np.array(df_duration['duration'])
```

In [10]:
```
#here we see there is no NaN in duration
np.any(np.isnan(df_duration['duration']))
```

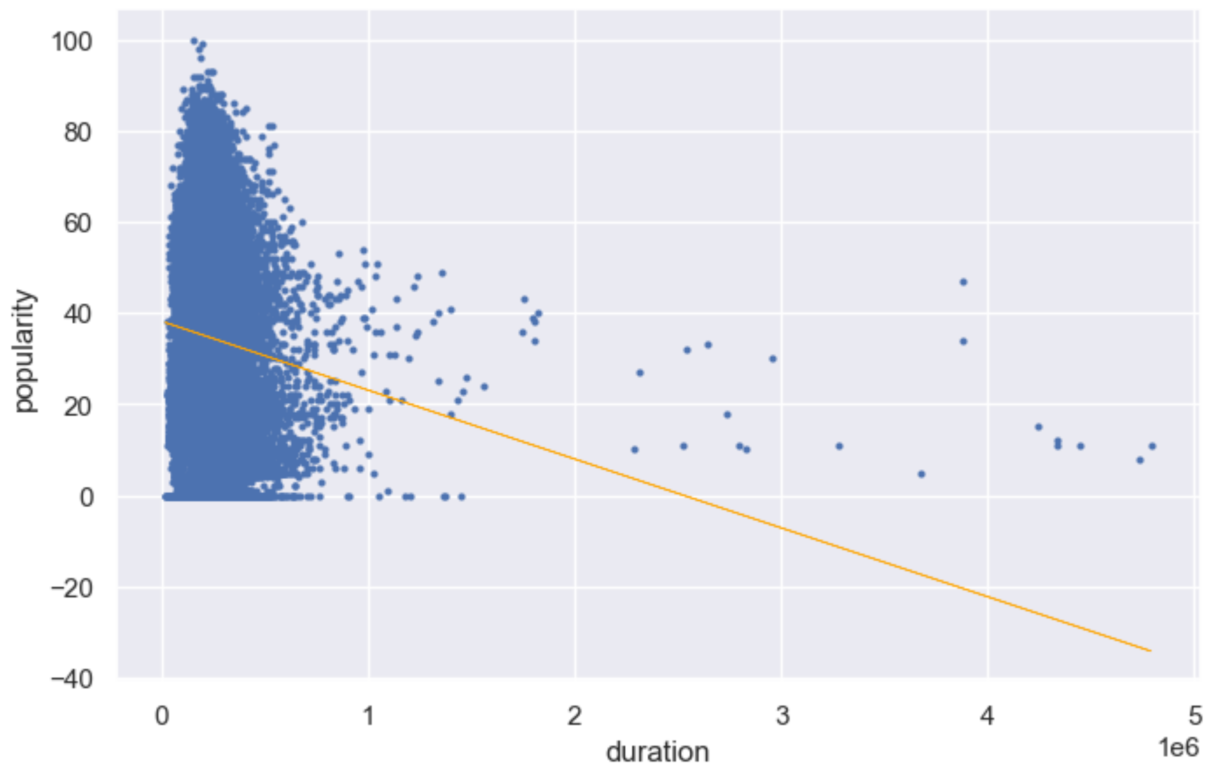
Out[10]: False

In [11]:
```
# PLotting a scatterplot between popularity and duration
plt.scatter(duration, popularity, s = 5)
plt.xlabel('duration')
plt.ylabel('popularity')
plt.show()
```



In [12]:
```
#from first glance seems longer duartion decreases popularity but hard to tell
```

In [13]:
```
#lets regress and see
reg = LinearRegression()
# we need to expand a dimension as Linear Regression takes input of shape (num
duration_exp = duration.reshape(-1,1) # going from shape (1000) -> (1000,1)
popularity = popularity.reshape(-1,1)
# fit our model on the data
reg.fit(duration_exp, popularity)

# Calculate y_hat
y_hat = reg.predict(duration_exp)

rmse_together = np.sqrt(np.mean((popularity-y_hat)**2))
# Plotting our ground truth and our predictions
plt.figure(figsize=(8,5))
plt.plot(duration_exp, popularity, 'o', ms=2)
plt.xlabel('duration')
plt.ylabel('popularity')
plt.plot(duration_exp, y_hat, color='orange', linewidth=0.5) # orange line for
plt.show()
```



In [14]:
```
rmse_together
```

Out[14]: 20.004676766057894

In [15]:
```
#ok well clearly this idnt accurate for the ones all the way right
```

In [16]:
```
#there seems to be a divide around 2,000,000 lets do two different regressions
```

In [17]:
```
#Lets see if this holds with correlation
correlation_coefficient = df_duration['duration'].corr(df_duration['popularity
```

In [18]:
```
correlation_coefficient
```

Out[18]: -0.09122431271265256

In [19]:
```
#good we want this to equal r2
correlation_coefficient**2
```

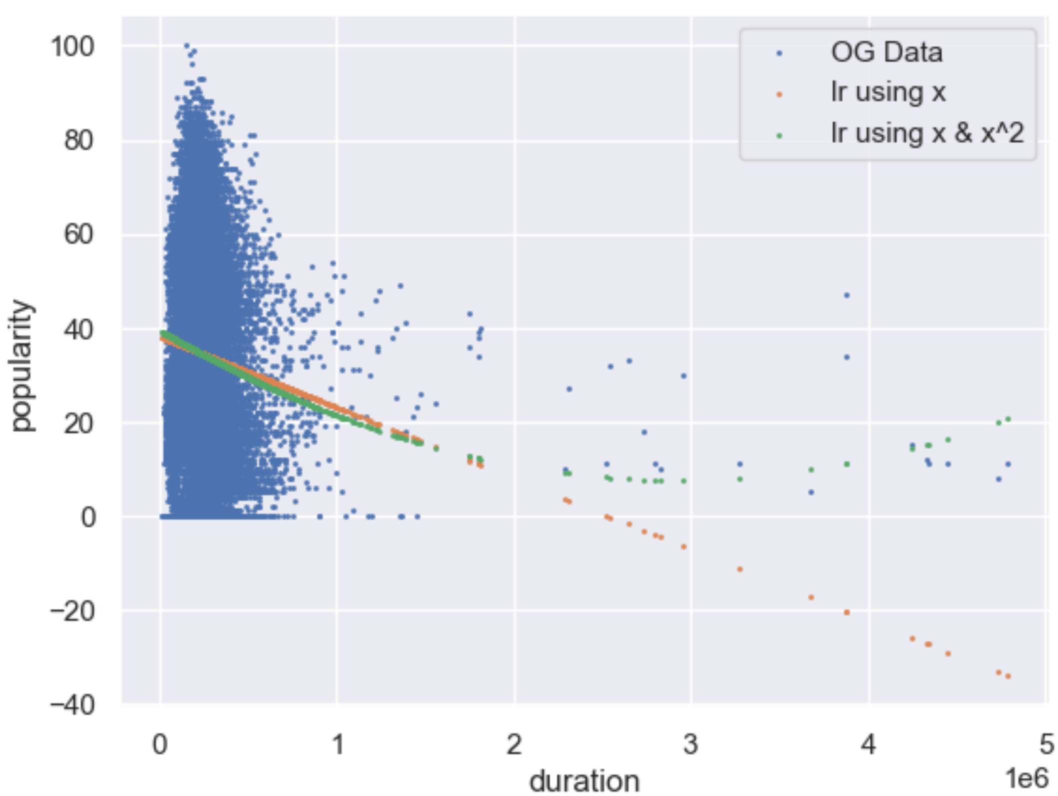
Out[19]: 0.008321875229895824

In [20]:
```
#slight negative correlation
```

In [21]:
```
rmse_single = np.sqrt(np.mean((popularity-y_hat)**2))
```

In [22]:
```
x = duration
y = popularity.reshape(-1,1)
x_sq = x*x
x_both = np.concatenate((x.reshape(-1,1),x_sq.reshape(-1,1)), axis = 1)
# print(x_both.shape)


# Model
reg_both = LinearRegression().fit(x_both, y)
y_hat_both = reg_both.predict(x_both.reshape(-1,2))

# RMSE Value
rmse_both = np.sqrt(np.mean((y-y_hat_both)**2))
print('RMSE earlier was: ', rmse_single)
print('RMSE now is: ', rmse_both)

# Plotting
plt.scatter(x,y, s = 1, label = 'OG Data')
plt.scatter(x,y_hat, s = 1, label = 'lr using x')
plt.scatter(x,y_hat_both, s = 1, label = 'lr using x & x^2')
plt.legend()
plt.xlabel('duration')
plt.ylabel('popularity')
plt.show()
```

```
RMSE earlier was:  20.004676766057894
RMSE now is:  19.990240318366386
```



In [23]:
```
#lets regress and see
reg = LinearRegression()
# we need to expand a dimension as Linear Regression takes input of shape (num
x = long_dur.reshape(-1,1) # going from shape (1000) -> (1000,1)
y = long_pop.reshape(-1,1)
# fit our model on the data
reg.fit(x, y)

# Calculate y_hat
y_hat = reg.predict(x)
rmse_short = np.sqrt(np.mean((y-y_hat)**2))


# Plotting our ground truth and our predictions
plt.figure(figsize=(8,5))
plt.plot(x, y, 'o', ms=2)
plt.xlabel('duration')
plt.ylabel('popularity')
plt.plot(x, y_hat, color='orange', linewidth=0.5) # orange line for the fox
plt.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[23], line 4
      2 reg = LinearRegression()
      3 # we need to expand a dimension as Linear Regression takes input of s
hape (num_samples, num_features)
----> 4 x = long_dur.reshape(-1,1) # going from shape (1000) -> (1000,1)
      5 y = long_pop.reshape(-1,1)
      6 # fit our model on the data

NameError: name 'long_dur' is not defined
```

In [ ]:
```
long_dur = df_duration[df_duration['duration'] >2000000]['duration'].values
```

In [ ]:
```
long_pop = df_duration[df_duration['duration'] >2000000]['popularity'].values
```

In [24]:
```
correlation_coefficient = df_duration[df_duration['duration'] >2000000]['durat
```

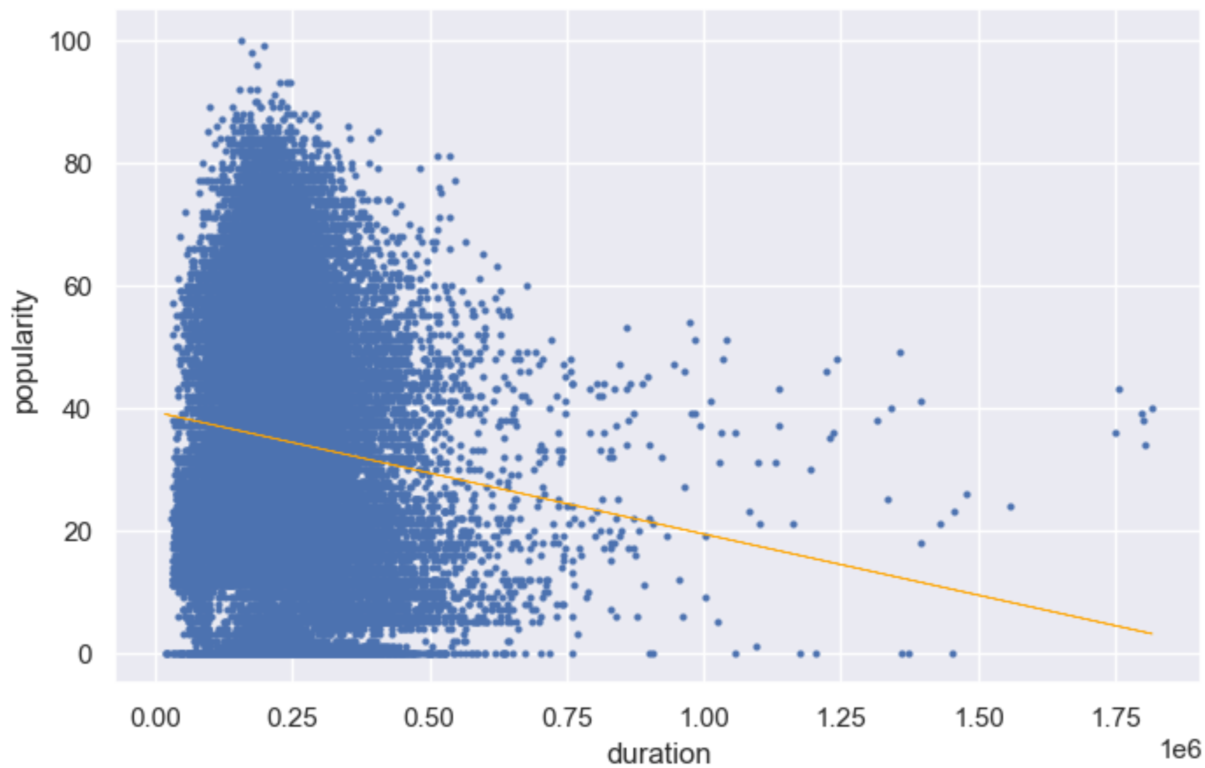
In [35]:
```
correlation_coefficient
```

Out[35]:
```
-0.2238066058341059
```

In [25]:
```
short_dur = df_duration[df_duration['duration'] <=2000000]['duration'].values
```

In [26]:
```
short_pop = df_duration[df_duration['duration'] <=2000000]['popularity'].value
```

In [31]:
```
#lets regress and see
reg = LinearRegression()
# we need to expand a dimension as Linear Regression takes input of shape (num
x = short_dur.reshape(-1,1) # going from shape (1000) -> (1000,1)
y = short_pop.reshape(-1,1)
# fit our model on the data
reg.fit(x, y)

# Calculate y_hat
y_hat = reg.predict(x)
rmse_long = np.sqrt(np.mean((y-y_hat)**2))

# Plotting our ground truth and our predictions
plt.figure(figsize=(8,5))
plt.plot(x, y, 'o', ms=2)
plt.xlabel('duration')
plt.ylabel('popularity')
plt.plot(x, y_hat, color='orange', linewidth=0.5) # orange line for the fox
plt.show()
```



In [36]:
```
correlation_coefficient = df_duration[df_duration['duration'] <=2000000]['dura
```

In [37]:
```
correlation_coefficient
```

Out[37]: -0.0990139636188038

Question 3 - Are songs in major key more popular than songs in minor key?

In [25]:
```
df_mode = df[['artists', 'track_name', 'popularity','mode']]
```

In [26]:
```
df_mode = df_mode.drop_duplicates()
```

In [27]:
```
#great no NaN keys
np.any(np.isnan(df_mode['mode']))
```

Out[27]:
```
False
```

In [28]:
```
#this isnt categorical data, we cant do parametric tests because popularity no
mode = np.array(df_mode['mode'])
```

In [29]:
```
major_pop = df_mode['popularity'][df_mode['mode'] == 1].values
```

In [30]:
```
minor_pop = df_mode['popularity'][df_mode['mode'] == 0].values
```

In [31]:
```
np.median(major_pop)
```

Out[31]:
```
34.0
```

In [32]:
```
np.median(minor_pop)
```

Out[32]:
```
35.0
```

In [33]:
```
major_pop.std()
```

Out[33]:
```
19.81813122895111
```

In [34]:
```
minor_pop.std()
```

Out[34]:
```
20.50865662611052
```

In [35]:
```
stats.mannwhitneyu(major_pop, minor_pop, alternative='greater')
```

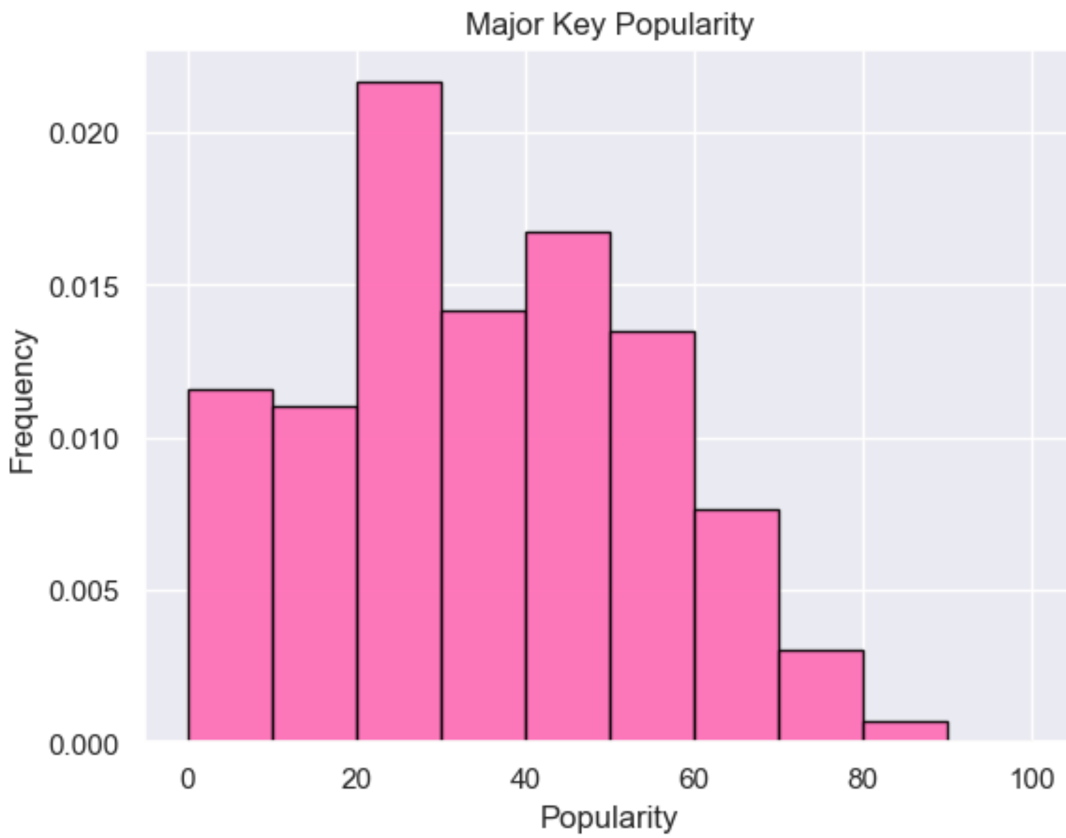
Out[35]:
```
MannwhitneyuResult(statistic=224859984.0, pvalue=0.9907374395848155)
```

In [52]:
```
stats.kstest(minor_pop, major_pop, alternative='two-sided')
```

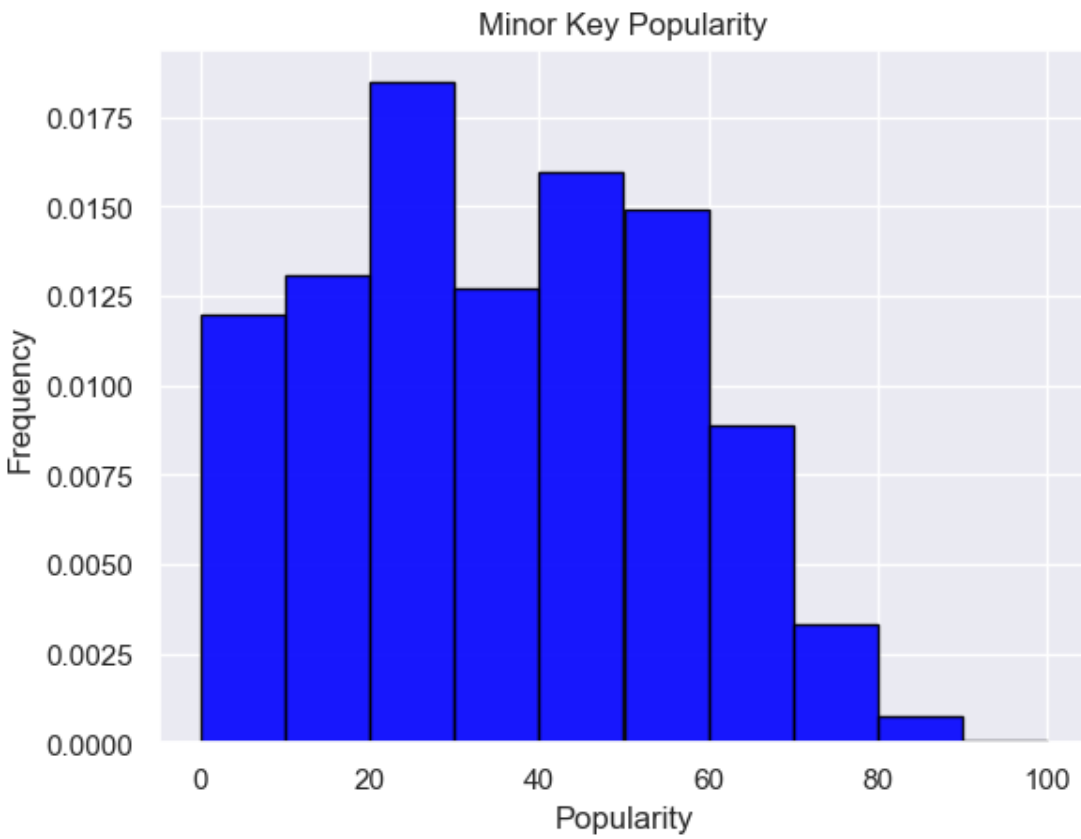
Out[52]:
```
KstestResult(statistic=0.03206174667866257, pvalue=1.1230601215328496e-09, st
atistic_location=47, statistic_sign=-1)
```

In [53]:
```
#popularity not normally distributed, thus cant use parametric test
```

In [57]:
```
plt.hist(major_pop, bins = [0,10,20,30,40,50,60,70,80,90,100],density = True,
plt.title('Major Key Popularity')
plt.xlabel('Popularity')
plt.ylabel('Frequency')
# plt.xlabel('Rating')
# plt.ylabel('Proportion of Ratings')
# plt.title('Ratings Distribution of Shrek (2001) by Females')

plt.show()
```



In [56]:
```
plt.hist(minor_pop, bins = [0,10,20,30,40,50,60,70,80,90,100],density = True,
plt.title('Minor Key Popularity')
plt.xlabel('Popularity')
plt.ylabel('Frequency')
# plt.title('Ratings Distribution of Shrek (2001) by Females')

plt.show()
```



Question 4 -Which of the following 10 song features: duration, danceability, energy, loudness, speechiness, acousticness, instrumentalness, liveness, valence and tempo predicts popularity best?

In [7]:
```
features = ["duration", "danceability", "energy", 'loudness', 'speechiness', '
```

In [8]:
```
features_plus = ["duration", "danceability", "energy", 'loudness', 'speechines
```

In [9]:
```
features_plus
```

Out[9]:
```
['duration',
 'danceability',
 'energy',
 'loudness',
 'speechiness',
 'acousticness',
 'instrumentalness',
 'liveness',
 'valence',
 'tempo',
 'popularity']
```

In [10]:
```
df_features = df[features_plus]
```

In [11]:
```
df_features = df_features.drop_duplicates()
```

In [12]:
```
df_features.shape
```

Out[12]:
```
(44088, 11)
```

In [23]:
```
models = {}
y = df_features['popularity'].values.reshape(-1,1)
for feature in features:
    X = df_features[feature].values.reshape(-1,1)
    x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r
    #since calculating slope of regression line squared can use train
    reg = LinearRegression().fit(x_train, y_train)
    y_hat = reg.predict(x_test)
    r2 = r2_score(y_test, y_hat)
    rmse = np.sqrt(np.mean(np.sum((y_test-y_hat)**2)))
    models[feature] = [r2, rmse]
```

In [24]:
```
models
```

Out[24]:
```
{'duration': [0.008193830340911035, 1875.9126901877396],
 'danceability': [0.005232011523969682, 1878.7116098849326],
 'energy': [0.007888235787609665, 1876.2016703161912],
 'loudness': [0.004085457987022978, 1879.7939844409432],
 'speechiness': [0.004991308407817585, 1878.938891216902],
 'acousticness': [0.001387547015716195, 1882.3384230386357],
 'instrumentalness': [0.04041955640069339, 1845.1849885166905],
 'liveness': [0.002725984425647976, 1881.0765536789343],
 'valence': [-5.755595695156046e-05, 1883.6999068935734],
 'tempo': [0.001264973536038827, 1882.453942172299]}
```

In [25]:
```
models_df = pd.DataFrame(models)
```

In [26]:
```
models_df.index = ['r2', 'RMSE']
```

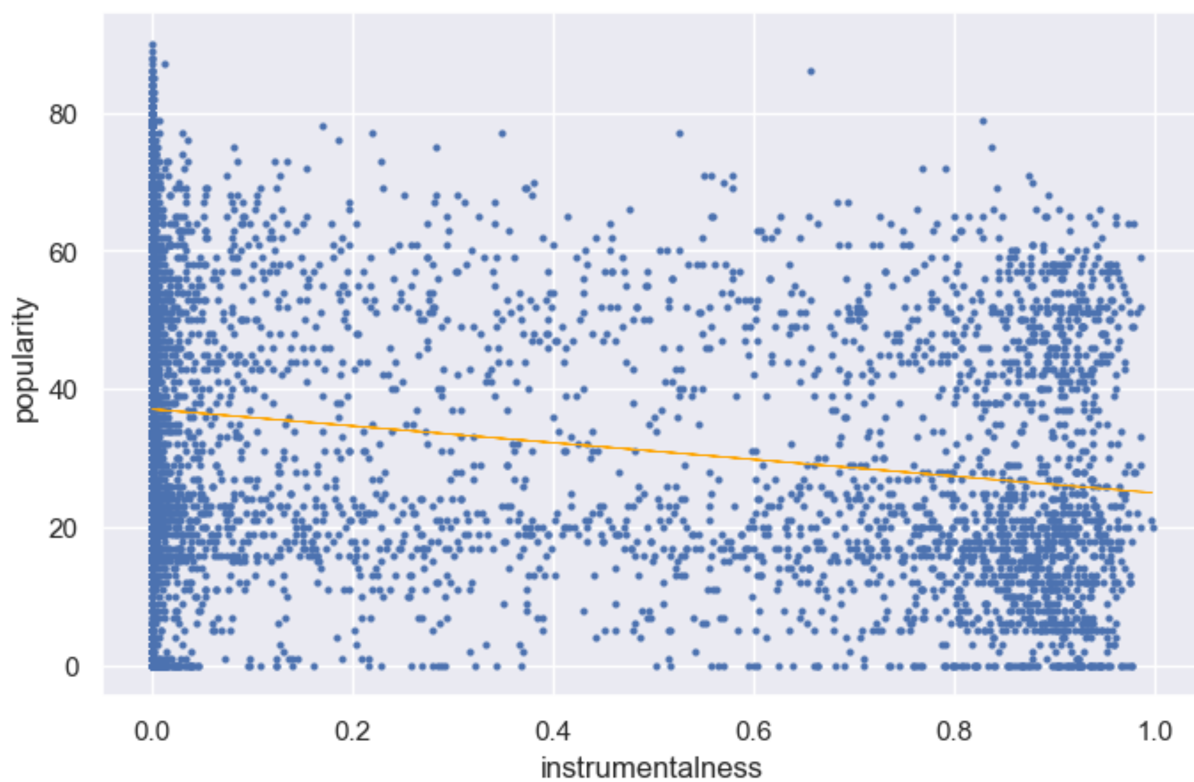
In [27]:
```
models_df
```

Out[27]:

	duration	danceability	energy	loudness	speechiness	acousticness	instrumentalness
r2	0.008194	0.005232	0.007888	0.004085	0.004991	0.001388	0.040420
RMSE	1875.912690	1878.711610	1876.201670	1879.793984	1878.938891	1882.338423	1845.184989

In [28]:
```
#instrumentalness has the lowest RMSE and the highest R^2, best predictor
# Plotting test
X = df_features['instrumentalness'].values.reshape(-1,1)
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando
reg = LinearRegression().fit(x_train, y_train)
print(reg.coef_)
y_hat = reg.predict(x_test)
r2 = r2_score(y_test,reg.predict(x_test))
rmse = np.sqrt(np.mean(np.sum((y_test-y_hat)**2)))

plt.figure(figsize=(8,5))
plt.plot(x_test, y_test, 'o', ms=2)
plt.ylabel('popularity')
plt.xlabel('instrumentalness')
plt.plot(x_test, reg.predict(x_test), color='orange', linewidth=0.5) # orange
plt.show()
```

```
[[-12.13008087]]
```



In [30]:
```
r2
```

Out[30]:
```
0.04041955640069339
```

In [42]:
```
#we see a negative correlation between instrumentalness and popularity
```

Question 6

In [159]:
```
features = ["duration", "danceability", "energy", 'loudness', 'speechiness', '
```

In [160]:
```
features.append('artists')
```

In [161]:
```
features.append('track_name')
```

In [162]:
```
features
```

Out[162]:
```
['duration',
 'danceability',
 'energy',
 'loudness',
 'speechiness',
 'acousticness',
 'instrumentalness',
 'liveness',
 'valence',
 'tempo',
 'artists',
 'track_name']
```

In [163]:
```
features
```

Out[163]:
```
['duration',
 'danceability',
 'energy',
 'loudness',
 'speechiness',
 'acousticness',
 'instrumentalness',
 'liveness',
 'valence',
 'tempo',
 'artists',
 'track_name']
```

In [164]:
```
df_6 = df[features]
```

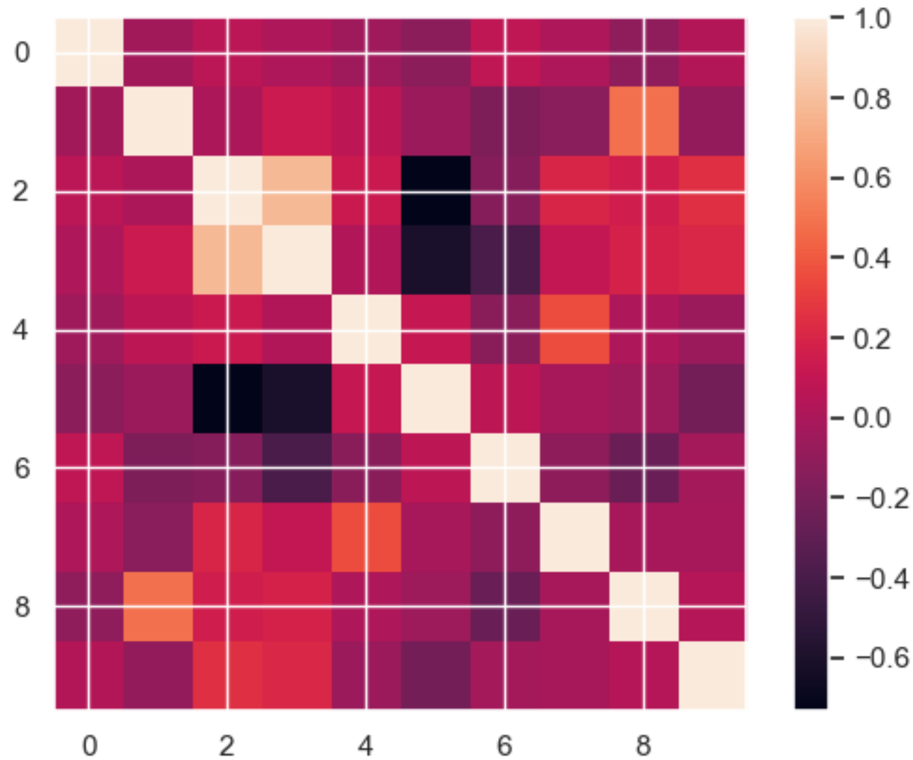
In [165]:
```
df_6 = df_6.drop_duplicates()
```

In [168]:
```
features = ["duration", "danceability", "energy", 'loudness', 'speechiness', '
```

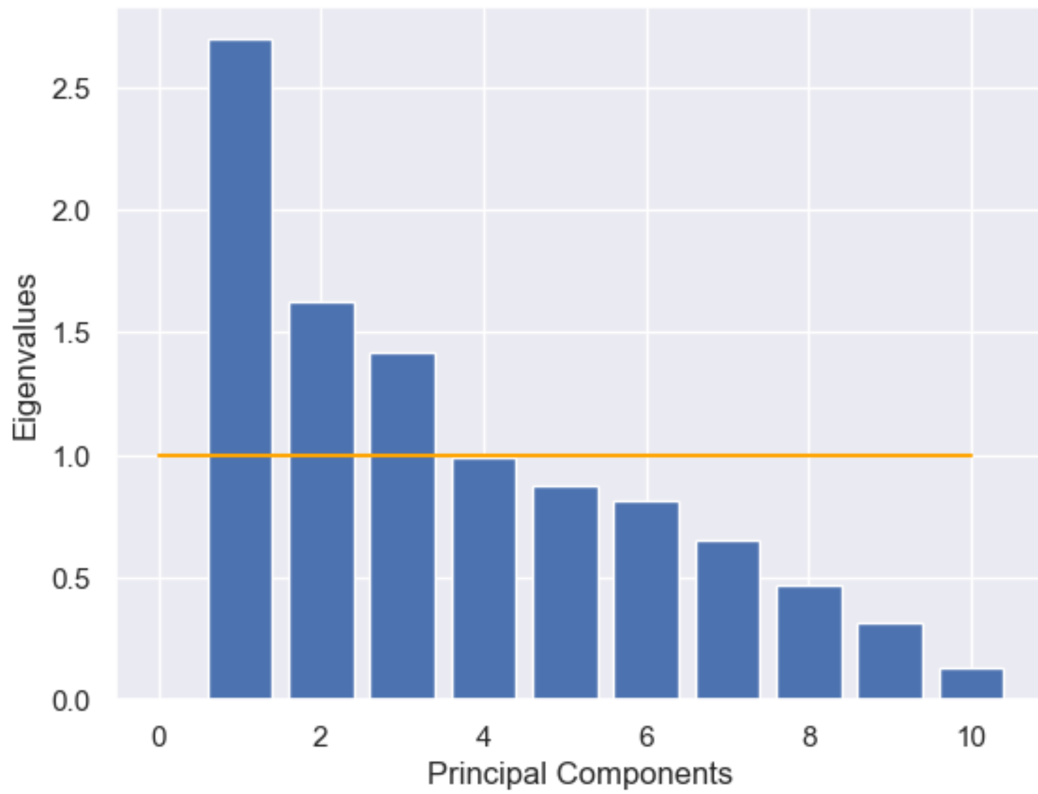
In [169]:
```
predictors = df_6[features].to_numpy()
```

In [170]:
```
# 4. Run the PCA

# Z-score the data:
zscoredData = stats.zscore(predictors)

# Initialize PCA object and fit to our data:
pca = sk_PCA().fit(zscoredData)

# Eigenvalues: Single vector of eigenvalues in decreasing order of magnitude
eigVals = pca.explained_variance_

# Loadings (eigenvectors): Weights per factor in terms of the original data.
loadings = pca.components_*-1

# Rotated Data - simply the transformed data:
origDataNewCoordinates = pca.fit_transform(zscoredData)*-1
```

In [171]:
```
# 3. Visualize correlation matrix
r = np.corrcoef(predictors,rowvar=False)
plt.imshow(r)
plt.colorbar()
plt.show()
```



In [187]:
```
# 5. Plot the eigenvalues
#Seems the cutoff would be around 3...? "Elbow method"
numPredictors = np.size(predictors,axis=1)
plt.bar(np.linspace(1,numPredictors,numPredictors),eigVals)
plt.xlabel('Principal Components')
plt.ylabel('Eigenvalues')
plt.plot([0,10],[1,1],color='orange') # Orange Kaiser criterion line for the f
plt.show()
print('Proportion variance explained by the first 3 PCs:',np.sum(eigVals[:3]/n
```




```
Proportion variance explained by the first 3 PCs: 0.575
```

In [188]:
```
X = np.column_stack((origDataNewCoordinates[:,0],origDataNewCoordinates[:,1],o
```

In [86]:
```
#pca_pipeline = Pipeline([('scaling', StandardScaler()), ('pca', sk_PCA())])
#predictors_processed = pca_pipeline.fit_transform(predictors)
```

In [87]:
```
X.shape
```

Out[87]:
```
(42933, 3)
```

In [88]:
```
X
```

Out[88]:
```
array([[ 0.36056933,  1.40261588, -0.38213032],
       [-3.54619081,  0.77656793, -0.36889647],
       [-1.3580977 , -0.35534212, -0.04199617],
       ...,
       [ 0.47271884,  1.78810938,  0.72241156],
       [ 1.05943015,  0.83216256, -0.44744986],
       [ 1.67730043,  1.09328498,  1.34925514]])
```

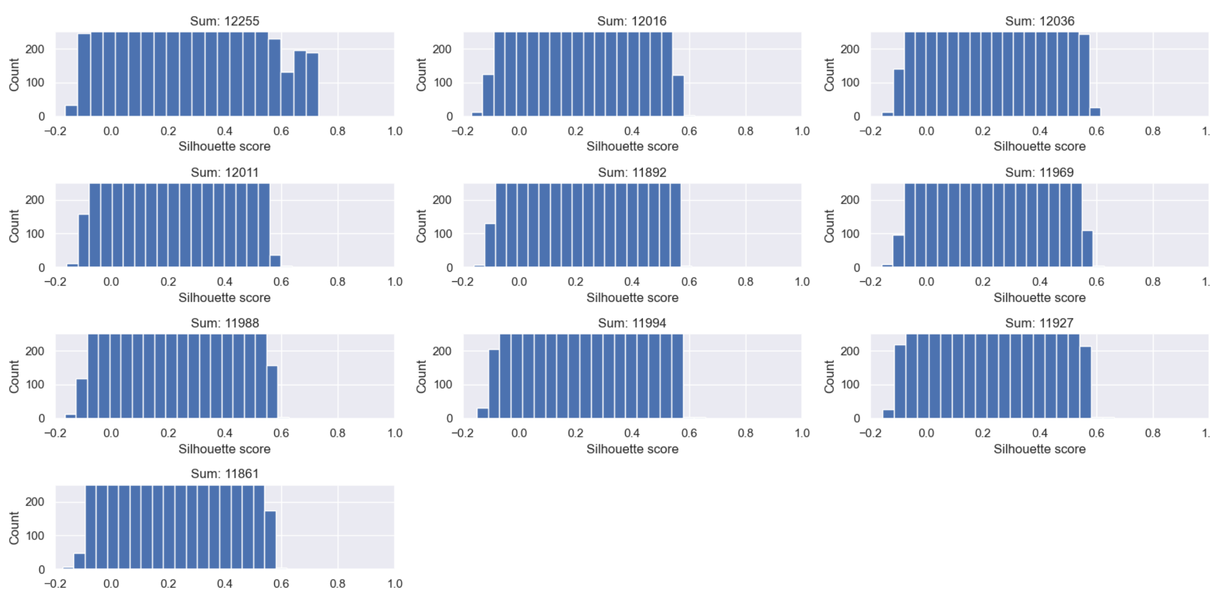
In [189]:
```
#wed expect this many different clusters
df['track_genre'].nunique()
```

Out[189]:
```
52
```

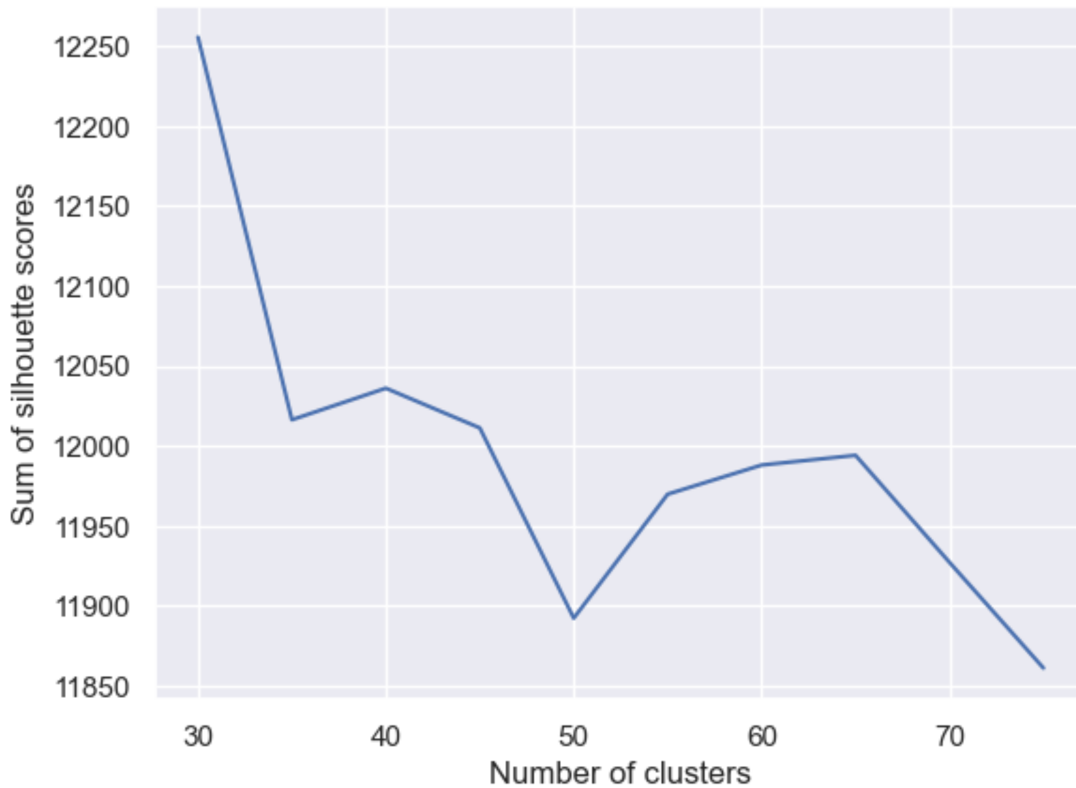
In [120]:
```
#so we should check around that range
#lets start with a more rough search in groups of 5
#will then narrow down
```

In [121]:
```
numClusters = 10 # how many clusters are we looping over? (from 2 to 10)
Q = np.empty([numClusters,1])*np.NaN # init container to store sums

# Compute kMeans:
plt.figure(figsize=(16, 8))
i = 0
for ii in [30,35,40,45,50,55,60,65,70,75]: # Loop through each cluster
    kMeans = sk_KMeans(n_clusters = int(ii), random_state=random).fit(X) # com
    cId = kMeans.labels_ # vector of cluster IDs that the row belongs to
    cCoords = kMeans.cluster_centers_ # coordinate location for center of each
    s = silhouette_samples(X,cId) # compute the mean silhouette coefficient of
    Q[i] = sum(s) # take the sum
    # Plot data:
    plt.subplot(4,3,i+1)
    plt.hist(s,bins=20)
    plt.xlim(-0.2,1)
    plt.ylim(0,250)
    plt.xlabel('Silhouette score')
    plt.ylabel('Count')
    plt.title('Sum: {}'.format(int(Q[i]))) # sum rounded to nearest integer
    plt.tight_layout() # adjusts subplot padding
    i = i+1
```

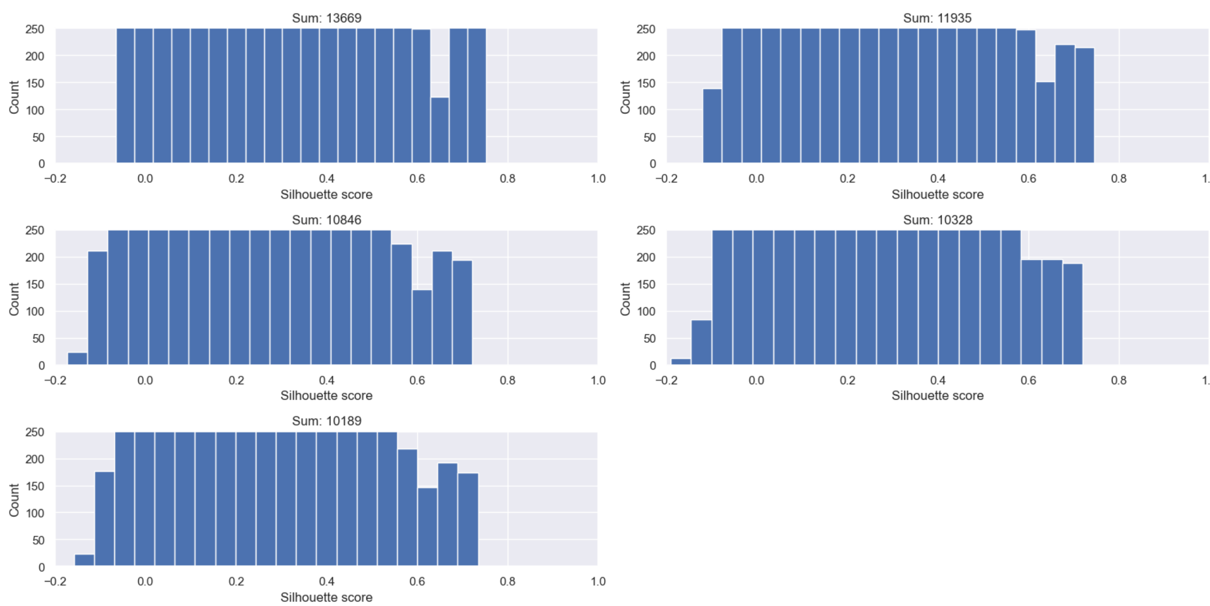
```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
File <__array_function__ internals>:177, in where(*args, **kwargs)

KeyboardInterrupt: 


Exception ignored in: 'sklearn.cluster._k_means_common._relocate_empty_cluste
rs_dense'
Traceback (most recent call last):
  File "<__array_function__ internals>", line 177, in where
KeyboardInterrupt: 
```



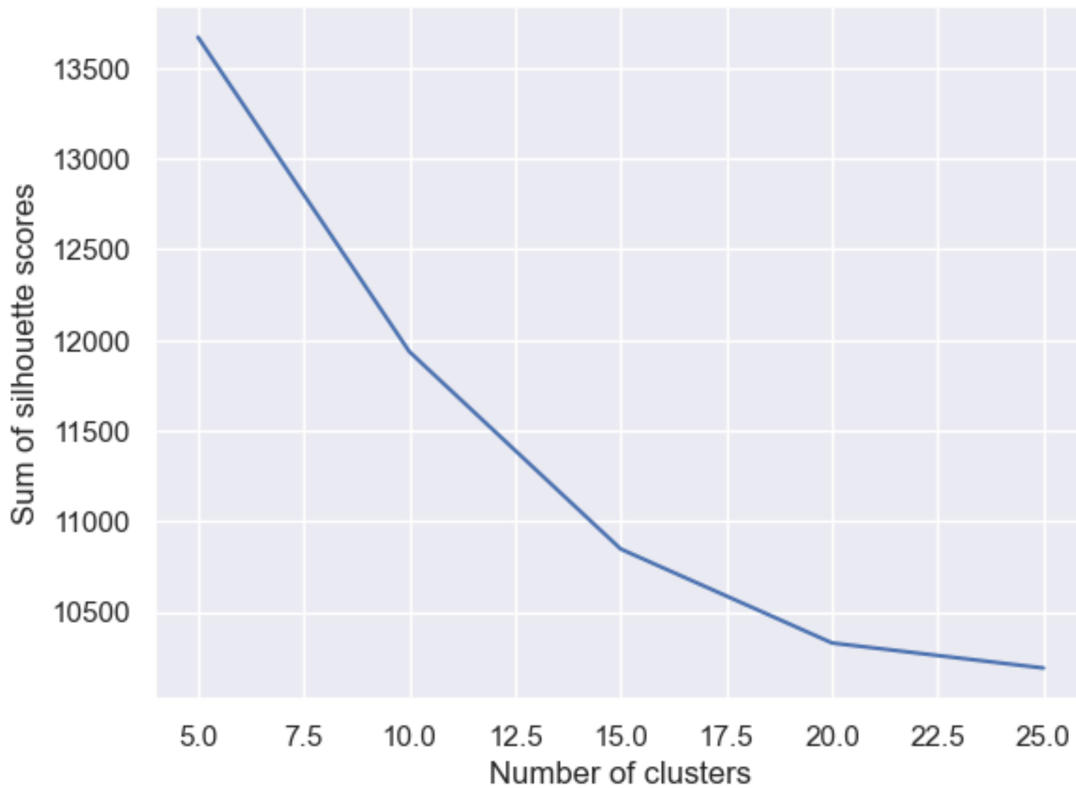
In [122]:
```
plt.plot(np.linspace(30,75,10),Q)
plt.xlabel('Number of clusters')
plt.ylabel('Sum of silhouette scores')
plt.show()
```



In [184]:
```
numClusters = 5 # how many clusters are we looping over? (from 2 to 10)
Q = np.empty([numClusters,1])*np.NaN # init container to store sums

# Compute kMeans:
plt.figure(figsize=(16, 8))
i = 0
for ii in [5,10,15,20,25]: # Loop through each cluster
    kMeans = sk_KMeans(n_clusters = int(ii), random_state=random).fit(X) # com
    cId = kMeans.labels_ # vector of cluster IDs that the row belongs to
    cCoords = kMeans.cluster_centers_ # coordinate location for center of each
    s = silhouette_samples(X,cId) # compute the mean silhouette coefficient of
    Q[i] = sum(s) # take the sum
    # Plot data:
    plt.subplot(3,2,i+1)
    plt.hist(s,bins=20)
    plt.xlim(-0.2,1)
    plt.ylim(0,250)
    plt.xlabel('Silhouette score')
    plt.ylabel('Count')
    plt.title('Sum: {}'.format(int(Q[i]))) # sum rounded to nearest integer
    plt.tight_layout() # adjusts subplot padding
    i = i+1
```

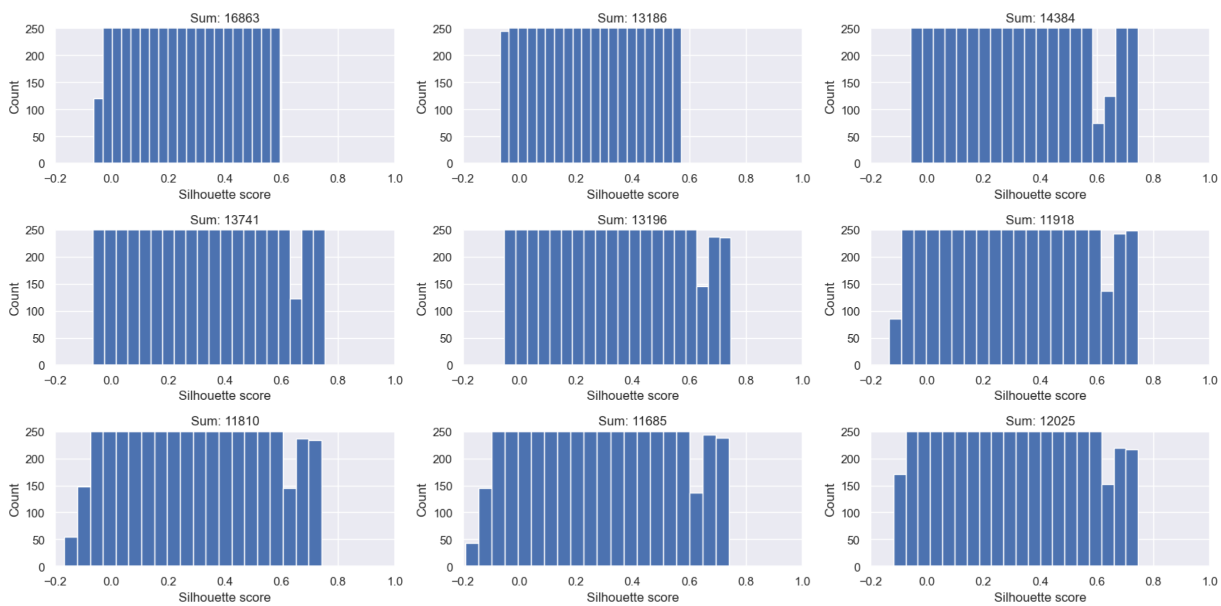


In [185]:
```
plt.plot(np.linspace(5,25,5),Q)
plt.xlabel('Number of clusters')
plt.ylabel('Sum of silhouette scores')
plt.show()
```

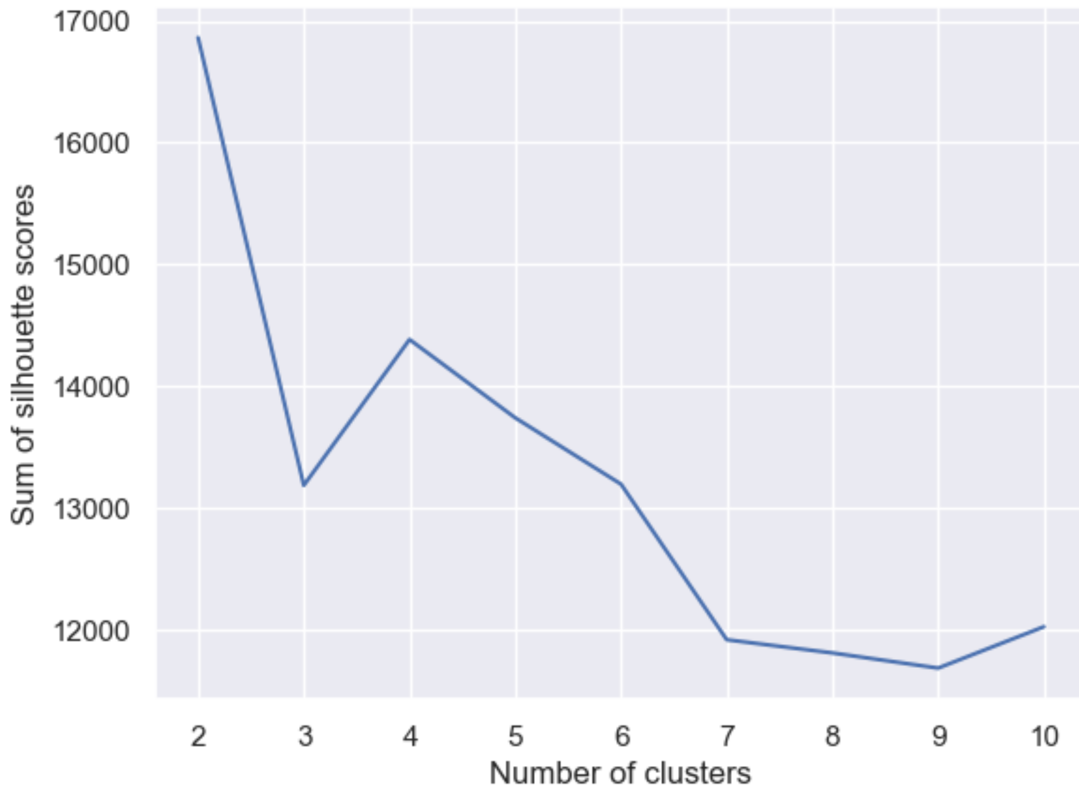


In [ ]:
```
#we go even more left
```

In [91]:
```
numClusters = 9 # how many clusters are we looping over? (from 2 to 10)
Q = np.empty([numClusters,1])*np.NaN # init container to store sums

# Compute kMeans:
plt.figure(figsize=(16, 8))
i = 0
for ii in [2,3,4,5,6,7,8,9,10]: # Loop through each cluster
    kMeans = sk_KMeans(n_clusters = int(ii), random_state=random).fit(X) # com
    cId = kMeans.labels_ # vector of cluster IDs that the row belongs to
    cCoords = kMeans.cluster_centers_ # coordinate location for center of each
    s = silhouette_samples(X,cId) # compute the mean silhouette coefficient of
    Q[i] = sum(s) # take the sum
    # Plot data:
    plt.subplot(3,3,i+1)
    plt.hist(s,bins=20)
    plt.xlim(-0.2,1)
    plt.ylim(0,250)
    plt.xlabel('Silhouette score')
    plt.ylabel('Count')
    plt.title('Sum: {}'.format(int(Q[i]))) # sum rounded to nearest integer
    plt.tight_layout() # adjusts subplot padding
    i = i+1
```



In [92]:
```
#using this fig to show we went for 2
plt.plot(np.linspace(2,10,9),Q)
plt.xlabel('Number of clusters')
plt.ylabel('Sum of silhouette scores')
plt.show()
```



In [ ]:
```
#best clustering is 2
```

In [190]:
```
kMeans = sk_KMeans(n_clusters = int(2), random_state=random).fit(X)
```

In [191]:
```
kMeans.labels_
```

Out[191]:
```
array([0, 1, 1, ..., 0, 0, 0], dtype=int32)
```

In [192]:
```
features = ["duration", "danceability", "energy", 'loudness', 'speechiness', '
```

In [193]:
```
df_k = df[features]
```

In [194]:
```
#ignore genre
df_k = df_k.drop_duplicates(subset=features[:-1])
```

In [195]:
```
kMeans.labels_.shape
```

Out[195]:
```
(42933,)
```

In [196]:
```
df_k.shape
```

Out[196]:
```
(42933, 13)
```

In [197]:
```
df_k['cluster'] = kMeans.labels_
```

In [198]:
```
df_k
```

Out[198]:

songNumber	duration	danceability	energy	loudness	speechiness	acousticness	instrumentalness	livene
0	230666	0.676	0.4610	-6.746	0.1430	0.03220	0.000001	0.35
1	149610	0.420	0.1660	-17.235	0.0763	0.92400	0.000006	0.10
2	210826	0.438	0.3590	-9.734	0.0557	0.21000	0.000000	0.11
3	201933	0.266	0.0596	-18.515	0.0363	0.90500	0.000071	0.13
4	198853	0.618	0.4430	-9.681	0.0526	0.46900	0.000000	0.08
...	...	...	...	...	...	...	...	
51993	233720	0.759	0.8850	-4.516	0.0636	0.35100	0.000037	0.46
51994	212413	0.831	0.8180	-7.827	0.0824	0.02450	0.000319	0.08
51995	203653	0.819	0.6450	-6.707	0.0481	0.23200	0.000863	0.17
51998	168620	0.727	0.6470	-7.383	0.2800	0.03290	0.000000	0.24
51999	232000	0.685	0.8620	-4.611	0.0627	0.00757	0.001400	0.02

42933 rows × 14 columns

In [199]:
```
genre_cluster_counts = df_k.groupby('track_genre')['cluster'].value_counts().u
```

In [200]:
```
genre_cluster_percentages = genre_cluster_counts.div(genre_cluster_counts.sum(
```

In [201]:
```
genre_cluster_percentages.columns
```

Out[201]:
```
Index([0, 1], dtype='int32', name='cluster')
```

In [202]:
```
genre_cluster_percentages[genre_cluster_percentages[1] > 50].sort_values(by=1,
```

Out[202]:

cluster	0	1
track_genre		
classical	7.894737	92.105263
ambient	10.395010	89.604990
disney	20.102041	79.897959
guitar	21.729238	78.270762
chill	42.361863	57.638137
acoustic	42.752868	57.247132
cantopop	47.076613	52.923387

In [205]:
```
df_export = genre_cluster_percentages[genre_cluster_percentages[0] > 50].sort_
```

In [206]:
```
df_export
```

Out[206]:

cluster	0	1
track_genre		
hardstyle	100.000000	0.000000
drum-and-bass	99.572193	0.427807
happy	99.383350	0.616650
hardcore	99.332443	0.667557
edm	99.270073	0.729927
death-metal	99.103139	0.896861
breakbeat	98.977505	1.022495
forro	98.478702	1.521298
heavy-metal	98.286290	1.713710
grindcore	98.170732	1.829268
dance	98.127341	1.872659
dubstep	96.887160	3.112840
grunge	96.801968	3.198032
dancehall	96.180556	3.819444
hard-rock	95.988113	4.011887
black-metal	95.262097	4.737903
deep-house	95.095949	4.904051
dub	95.005549	4.994451
hip-hop	94.768311	5.231689
chicago-house	94.619289	5.380711
alt-rock	92.712067	7.287933
funk	91.482650	8.517350
disco	90.284360	9.715640
alternative	89.959839	10.040161
goth	88.976378	11.023622
groove	88.562092	11.437908
club	87.487073	12.512927
afrobeat	87.436677	12.563323
emo	87.162162	12.837838
garage	85.891648	14.108352
electronic	85.000000	15.000000
detroit-techno	82.875264	17.124736
brazil	81.028939	18.971061
electro	80.382775	19.617225
country	78.929766	21.070234
blues	75.207756	24.792244

cluster	0	1
track_genre		
gospel	74.664680	25.335320
anime	74.186992	25.813008
french	71.641791	28.358209
german	65.378671	34.621329
folk	61.176471	38.823529
children	60.569551	39.430449
bluegrass	59.210526	40.789474
comedy	55.757576	44.242424
british	53.049482	46.950518

In [132]:
```
sum(genre_cluster_counts[0])/(sum(genre_cluster_counts[0])+sum(genre_cluster_c
```
Out[132]: 0.7776535532108169

In [131]:
```
sum(genre_cluster_counts[1])
```
Out[131]: 9546

In [215]:
```
df_k = df_k.drop(['artists', 'track_name', 'track_genre'], axis = 1)
```

In [230]:
```
average_by_label = df_k.groupby('cluster').median()
```

In [231]:
```
#its actually median
average_by_label
```
Out[231]:

	duration	danceability	energy	loudness	speechiness	acousticness	instrumentalness	liveness
cluster								
0	218997.0	0.588	0.7950	-5.958	0.0579	0.0416	0.000127	0.144
1	197682.5	0.519	0.3105	-13.076	0.0401	0.8090	0.009360	0.114

In [232]:
```
cluster_0 = df_k[df_k['cluster'] == 0]
cluster_1 = df_k[df_k['cluster'] == 1]
```

In [233]:
```
results_df = pd.DataFrame(index=['U-statistic', 'P-value'], columns=average_by_

for feature in average_by_label.columns:
    # Perform Mann-Whitney U test
    stat, p_value = stats.mannwhitneyu(cluster_0[feature], cluster_1[feature])

    # Display the results
    print(f"Mann-Whitney U test for {feature}:")
    print(f"  U-statistic: {stat}")
    print(f"  P-value: {p_value}")
    results_df.at['U-statistic', feature] = stat
    results_df.at['P-value', feature] = p_value
    # Check for significance (adjust the alpha level as needed)
    alpha = 0.05
    if p_value < alpha:
        print("  The difference is statistically significant.")
    else:
        print("  The difference is not statistically significant.")
    print("\n")
```

```
Mann-Whitney U test for duration:
  U-statistic: 189148299.0
  P-value: 2.6274958759478363e-166
  The difference is statistically significant.


Mann-Whitney U test for danceability:
  U-statistic: 195240569.0
  P-value: 1.8715317025502348e-241
  The difference is statistically significant.


Mann-Whitney U test for energy:
  U-statistic: 309395563.5
  P-value: 0.0
  The difference is statistically significant.


Mann-Whitney U test for loudness:
```

In [234]:
```
pd.options.display.float_format = None
```

In [235]:
```
results_df
```

Out[235]:

	duration	danceability	energy	loudness	speechiness	acousticness	instrumentalness
U-statistic	189148299.0	195240569.0	309395563.5	300740686.0	209686642.0	19205128.5	123182773.(
P-value	0.0	0.0	0.0	0.0	0.0	0.0	0.(

In [236]:
```
results_df2 = pd.concat([average_by_label, results_df], ignore_index=True)
```

In [237]:
```
results_df2.loc['P-value'] = results_df.loc['P-value'].apply(lambda x: f'{x:.2
```

In [238]:
```
results_df2 = results_df2.drop(3)
```

In [239]:
```
results_df2.index = ['Cluster 0', 'Cluster 1', 'U-value', 'P-value']
```

In [240]:
```
results_df2
```

Out[240]:

	duration	danceability	energy	loudness	speechiness	acousticness	instrumentalness
Cluster 0	218997.0	0.588	0.795	-5.958	0.0579	0.0416	0.000127
Cluster 1	197682.5	0.519	0.3105	-13.076	0.0401	0.809	0.00936
U-value	189148299.0	195240569.0	309395563.5	300740686.0	209686642.0	19205128.5	123182773.0
P-value	2.63e-166	1.87e-241	0.00e+00	0.00e+00	0.00e+00	0.00e+00	4.54e-263

In [ ]:
```
#maybe if were iffy about this we can not include it
```

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import pandas as pd
spotify_data = pd.read_csv('spotify52kData.csv')
from scipy.stats import ttest_ind
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import mannwhitneyu
spotify_data.set_index('songNumber', inplace=True)
spotify_data = spotify_data.drop_duplicates()
spotify_data
#Tyler Perez Question 2


# In[12]:




spotify_data = pd.read_csv('spotify52kData.csv')
spotify_data.set_index('songNumber', inplace=True)

columns_to_check = ['artists', 'album_name', 'track_name', 'explicit',
'popularity']


spotify_data = spotify_data.drop_duplicates(subset=columns_to_check)

spotify_data.reset_index(inplace=True)

spotify_data


# In[5]:


explicit= spotify_data[spotify_data['explicit'] == True]  # Select
rows where 'explicit' is True
non_explicit= spotify_data[spotify_data['explicit'] == False]  #
Select rows where 'explicit' is Fals
explicit_median = explicit['popularity'].median()
non_explicit_median = non_explicit['popularity'].median()


# In[6]:
```

```
variance_explicit = explicit['popularity'].var()
variance_non_explicit = non_explicit['popularity'].var()
avg_explicit = explicit['popularity'].mean()
avg_non_explicit = non_explicit['popularity'].mean()


# In[7]:


variance_explicit


# In[8]:


variance_non_explicit


# In[9]:


explicit_median


# In[10]:


non_explicit_median


# In[14]:


non_explicit


# In[64]:





# In[65]:



sns.set(style="whitegrid")


fig, axes = plt.subplots(1, 2, figsize=(12, 6), sharey=True)
```

```
# Plot the distribution of 'popularity' in explicit_df
sns.distplot(explicit['popularity'], bins=20, kde=True, color='blue',
ax=axes[0])
axes[0].set_title('Popularity Distribution (Explicit)')
axes[0].set_xlabel('Popularity')
axes[0].set_ylabel('Frequency')

# Plot the distribution of 'popularity' in non_explicit_df
sns.distplot(non_explicit['popularity'], bins=20, kde=True,
color='green', ax=axes[1])
axes[1].set_title('Popularity Distribution (Non-Explicit)')
axes[1].set_xlabel('Popularity')
axes[1].set_ylabel('Frequency')

# Adjust layout
plt.tight_layout()

# Show the plot
plt.show()


# In[15]:


statistic, p_value_mw = mannwhitneyu(explicit['popularity'],
non_explicit['popularity'])


# In[17]:


p_value_mw
statistic


# In[18]:


from scipy.stats import t

explicit_mean = explicit['popularity'].mean()
explicit_std = explicit['popularity'].std()
explicit_size = len(explicit)

explicit_conf_interval = t.interval(0.95, df=explicit_size-1,
loc=explicit_mean, scale=explicit_std / (explicit_size**0.5))

print("95% Confidence Interval for 'explicit' DataFrame -
Popularity:")
```

```
print(f"Mean: {explicit_mean}")
print("Confidence Interval:", explicit_conf_interval)

non_explicit_mean = non_explicit['popularity'].mean()
non_explicit_std = non_explicit['popularity'].std()
non_explicit_size = len(non_explicit)

non_explicit_conf_interval = t.interval(0.95, df=non_explicit_size-1,
loc=non_explicit_mean, scale=non_explicit_std /
(non_explicit_size**0.5))

print("\n95% Confidence Interval for 'non_explicit' DataFrame -
Popularity:")
print(f"Mean: {non_explicit_mean}")
print("Confidence Interval:", non_explicit_conf_interval)


# In[42]:




# In[35]:




# In[43]:




# In[44]:




# In[47]:




# In[ ]:
```

```
#!/usr/bin/env python
# coding: utf-8

# In[26]:


#Tyler Perez Question #8

import pandas as pd
from scipy.stats import ttest_ind, mannwhitneyu
import matplotlib.pyplot as plt
import seaborn as sns


spotify_data = pd.read_csv('spotify52kData.csv')

spotify_data.set_index('songNumber', inplace=True)

spotify_data = spotify_data.drop_duplicates()
spotify_data = spotify_data.drop_duplicates(subset=['artists',
'track_genre', "track_name", 'duration', 'danceability', 'energy',
'loudness', 'speechiness',
                  'acousticness', 'instrumentalness', 'liveness',
'valence', 'tempo'])


# In[2]:





# In[27]:


from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
#had help from chat gpt to get the format right for the neural network

#choosing our 10 features
X = spotify_data[['duration', 'danceability', 'energy', 'loudness',
'speechiness',
                  'acousticness', 'instrumentalness', 'liveness',
'valence', 'tempo']]
y = spotify_data['track_genre']
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=13839901)
#standarize features normalyl
```

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Use ReLU activation function
model = MLPClassifier(hidden_layer_sizes=(50, 25), activation='relu',
max_iter=1000, random_state=13839901)

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')


# In[28]:


from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt


y_pred = model.predict(X_test_scaled)


cm = confusion_matrix(y_test, y_pred)


plt.figure(figsize=(12, 8))

# Plot the confusion matrix using seaborn's heatmap
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=spotify_data['track_genre'].unique(),
yticklabels=spotify_data['track_genre'].unique())
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()


# In[4]:


from sklearn.metrics import classification_report

classification_rep = classification_report(y_test, y_pred,
target_names=spotify_data['track_genre'].unique())
```

```
print("Classification Report:\n", classification_rep)

# In[14]:




# In[7]:




# In[15]:




# In[16]:




# In[14]:




# In[18]:




# In[25]:




# In[30]:


import pandas as pd
from sklearn.decomposition import PCA
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X = spotify_data[['duration', 'danceability', 'energy', 'loudness',
'speechiness',
                  'acousticness', 'instrumentalness', 'liveness',
'valence', 'tempo']]
y = spotify_data['track_genre']

pca = PCA().fit(X)

eigVals = pca.explained_variance_

loadings = pca.components_

num_components = 3


X_pca = pca.transform(X)[:, :num_components]


scaler = StandardScaler()
X_pca_zscored = scaler.fit_transform(X_pca)


X_train, X_test, y_train, y_test = train_test_split(X_pca_zscored, y,
test_size=0.2, random_state=13839901)

model = MLPClassifier(hidden_layer_sizes=(50, 25), activation='relu',
max_iter=1000, random_state=13839901)


model.fit(X_train, y_train)

y_pred = model.predict(X_test)


accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')


# In[31]:


from sklearn.metrics import precision_score
```

```
y_pred = model.predict(X_test)


precision_scores = precision_score(y_test, y_pred, average=None,
labels=spotify_data['track_genre'].unique())

for i, genre in enumerate(spotify_data['track_genre'].unique()):
    print(f'Precision for {genre}: {precision_scores[i]:.2f}')


# In[ ]:
```

```
#!/usr/bin/env python
# coding: utf-8

# In[3]:


import pandas as pd
import numpy as np
import random

df = pd.read_csv('spotify52kData.csv')
ratings = pd.read_csv('starRatings.csv')

#Question #9 Tyler Perez


# In[4]:



ratings.columns.values[0] = "0"
ratings.columns = ratings.columns.str.replace('[a-zA-Z,.\'"\[\]{}()!
@#$%^&*;:]', '', regex=True)


# In[5]:


ratings


# In[5]:


col_means = ratings.mean()
row_means = ratings.mean(axis=1)
ratings_filled = ratings.apply(lambda col: col.fillna((col.mean() +
row_means) / 2), axis=0)
ratings=ratings_filled

df
# In[6]:


ratings_transposed = ratings.transpose()
ratings_transposed


# In[7]:
```

```
ratings_transposed['Average'] = ratings_transposed.mean(axis=1)


# In[19]:


ratings_transposed


# In[8]:


ratings_transposed_reset = ratings_transposed.reset_index()

df['Average'] = ratings_transposed_reset['Average']

result_df = df[['songNumber', 'artists', 'track_name', 'popularity',
'Average']]


# In[45]:


df


# In[9]:


result_df = result_df.dropna(subset=['Average'])
result_df
result_df = result_df.drop_duplicates(subset=['artists',
'track_name'], keep='first')
result_df


# In[9]:


correlation = result_df['popularity'].corr(result_df['Average'])
print(f"Correlation between 'popularity' and 'Average':
{correlation}")


# In[15]:


from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

```
from sklearn.metrics import mean_squared_error
import numpy as np


X = result_df[['popularity']]
y = result_df['Average']

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=13839901)


model = LinearRegression()


model.fit(X_train, y_train)

predictions = model.predict(X_test)

mse = mean_squared_error(y_test, predictions)


# In[16]:


standard_error = np.sqrt(np.sum((predictions - y_test) ** 2) /
(len(y_test) - 2)) / np.sqrt(np.sum((X_test['popularity'] -
np.mean(X_test['popularity'])) ** 2))


t_statistic = (model.coef_[0] - 0) / standard_error

print(f"t-statistic for 'popularity': {t_statistic}")

from scipy.stats import t

df = len(y_test) - 2


p_value = 2 * (1 - t.cdf(np.abs(t_statistic), df=df))

print(f"P-value for 'popularity': {p_value}")
print(f"Mean Squared Error: {mse}")
print("Regression Coefficients:")
print(f"Intercept: {model.intercept_}")
print(f"Coefficient for 'popularity': {model.coef_[0]}")
cod = model.score(X_test[['popularity']], y_test)

print(f"Coefficient of Determination (COD): {cod}")
```

```
# In[17]:


import matplotlib.pyplot as plt

plt.scatter(X_test['popularity'], y_test, label='Observed Data')

plt.plot(X_test['popularity'], predictions, color='red', linewidth=2,
label='Regression Line')

plt.title('Least Squares Linear Regression')
plt.xlabel('Popularity')
plt.ylabel('Average Rating')
plt.legend()
plt.grid(True)

plt.show()


# In[18]:


top_10_songs = result_df.nlargest(10, 'Average')
print(top_10_songs[['songNumber', 'artists','track_name',
'popularity', 'Average']])


# In[10]:


import scipy.stats

spearman_corr, _ = scipy.stats.spearmanr(result_df['popularity'],
result_df['Average'])

print("Spearman's correlation:", spearman_corr)


# In[ ]:
```

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:


import pandas as pd
spotify_data = pd.read_csv('spotify52kData.csv')
from scipy.stats import ttest_ind
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import mannwhitneyu
spotify_data.set_index('songNumber', inplace=True)
spotify_data = spotify_data.drop_duplicates()
spotify_data.drop_duplicates(subset=['track_name', 'popularity'],
inplace=True)
#Tyler Perez Question EC


# In[3]:


# Calculate the median length of track_name
median_length = spotify_data['track_name'].apply(len).median()

# Create two dataframes based on the median length
long_songs = spotify_data[spotify_data['track_name'].apply(len) >
median_length]
short_songs = spotify_data[spotify_data['track_name'].apply(len) <=
median_length]
median_length


# In[3]:



variance_long_songs = long_songs['popularity'].var()


variance_short_songs = short_songs['popularity'].var()


print("Variance of popularity for long_songs:", variance_long_songs)
print("Variance of popularity for short_songs:", variance_short_songs)


# In[5]:
```

```
# In[4]:


# Calculate mean and median popularity for long songs
mean_popularity_long = long_songs['popularity'].mean()
median_popularity_long = long_songs['popularity'].median()

# Calculate mean and median popularity for short songs
mean_popularity_short = short_songs['popularity'].mean()
median_popularity_short = short_songs['popularity'].median()
mean_popularity_short
mean_popularity_long


# In[5]:


# Calculate mean and median popularity for long songs
median_popularity_long
median_popularity_short
median_popularity_long


# In[6]:


from scipy.stats import mannwhitneyu

# Perform Mann-Whitney U test
statistic, p_value = mannwhitneyu(long_songs['popularity'],
short_songs['popularity'])


# In[7]:


p_value
statistic


# In[17]:


import seaborn as sns
import matplotlib.pyplot as plt

sns.set(style="whitegrid")
```

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(14, 6))

# Plot the distribution of popularity for Long Songs
sns.distplot(long_songs['popularity'], kde=True, color='blue',
ax=axes[0])
axes[0].set_title('Distribution of Popularity for Long Titled Songs')
axes[0].set_xlabel('Popularity')
axes[0].set_ylabel('Frequency')

# Plot the distribution of popularity for Short Songs
sns.distplot(short_songs['popularity'], kde=True, color='orange',
ax=axes[1])
axes[1].set_title('Distribution of Popularity for Short Titled Songs')
axes[1].set_xlabel('Popularity')
axes[1].set_ylabel('Frequency')

plt.tight_layout()


plt.show()


# In[16]:


import seaborn as sns
import matplotlib.pyplot as plt


sns.set(style="whitegrid")


fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(14, 6))


sns.boxplot(y='popularity', data=long_songs, ax=axes[0], color='blue')
axes[0].set_title('Box and Whisker Plot for Popularity - Long Titled
Songs')
axes[0].set_ylabel('Popularity')


sns.boxplot(y='popularity', data=short_songs, ax=axes[1],
color='orange')
axes[1].set_title('Box and Whisker Plot for Popularity - Short Titled
Songs')
axes[1].set_ylabel('Popularity')


plt.tight_layout()
```

```
plt.show()


# In[20]:




# In[8]:




n1 = len(long_songs['popularity'])
n2 = len(short_songs['popularity'])


degrees_of_freedom = n1 + n2 - 2

print(f"Degrees of Freedom: {degrees_of_freedom}")


# In[6]:


import pandas as pd
spotify_data.reset_index(inplace=True)

correlation =
spotify_data['songNumber'].corr(spotify_data['popularity'])

print(f"Correlation between songNumber and popularity: {correlation}")


# In[4]:




quantile_threshold = 0.99
upper_threshold =
long_songs['popularity'].quantile(quantile_threshold)
long_songs_no_outliers = long_songs[long_songs['popularity'] <=
upper_threshold]


statistic, p_value =
```

```
mannwhitneyu(long_songs_no_outliers['popularity'],
short_songs['popularity'])

print(f"Mann-Whitney U Statistic: {statistic}")
print(f"P-value: {p_value}")

if p_value < 0.05:
    print("The difference is statistically significant.")
else:
    print("The difference is not statistically significant.")


# In[ ]:
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Dec  5 11:24:35 2023

@author: tanvibansal
Capstone Q5
"""


import pandas as pd
import random
import numpy as np
from sklearn.linear_model import LinearRegression
from matplotlib import pyplot as plt
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

random.seed(13839901)
df_raw = pd.read_csv('spotify52kData.csv')
df = df_raw[['artists','track_name', 'popularity', 'duration', 'danceability', 'energy',
'loudness', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', 'valence' ,
'tempo']]
df = df.drop_duplicates()
#ratings = pd.read_csv('starRatings.csv')

#extract the features of interest for the question
features = ['duration', 'danceability', 'energy', 'loudness', 'speechiness', 'acousticness',
'instrumentalness', 'liveness', 'valence' , 'tempo']
y = df['popularity']
x = df[['duration', 'danceability', 'energy', 'loudness', 'speechiness', 'acousticness',
'instrumentalness', 'liveness', 'valence' , 'tempo']]

y = np.array(y)
x = np.array(x)
#split data set into testing/training sets for cross validation
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
random_state=random.seed(13839901))


#fit (unregularized) multiple regression model to training set
reg = LinearRegression()
reg.fit(x_train, y_train)

# compute y_hat
y_hat = reg.predict(x_test)

#Calculate R2 and RMSE
r2 = r2_score(y_test,y_hat)
rmse = np.sqrt(np.mean(np.sum((y_test-y_hat)**2)))


#fit regularized multiple regression model to training set for each alpha of interest
#collect R2 and RMSE for training and testing models
alpha = [1e-6, 1e-5, .0001, .001, .01, .1, 1, 10, 100, 1000, 10000]
reg_r2_train = []
reg_r2_test = []

reg_rmse_train = []
reg_rmse_test = []

for a in alpha:
    r_reg = Lasso(alpha=a)
    r_reg.fit(x_train, y_train)

    #compute y hat
```

```
    r_y_hat_train = r_reg.predict(x_train)
    r_y_hat_test = r_reg.predict(x_test)

    #calculate R2 and RMSE
    r_r2_train = r2_score(y_train,r_reg.predict(x_train))
    r_rmse_train = np.sqrt(np.mean(np.sum((y_train-r_y_hat_train)**2)))

    r_r2_test = r2_score(y_test,r_y_hat_test)
    r_rmse_test = np.sqrt(np.mean(np.sum((y_test-r_y_hat_test)**2)))

   #append r2 and RMSE to list for each value of alpha
    reg_r2_train.append(r_r2_train)
    reg_rmse_train.append(r_rmse_train)

    reg_r2_test.append(r_r2_test)
    reg_rmse_test.append(r_rmse_test)


#Plot training and testing RMSE as a function of regularization strength

fig, ax1 = plt.subplots()

color = 'tab:orange'
ax1.set_xlabel('Regularization Strength [lambda]')
ax1.set_ylabel('Train RMSE', color=color)
ax1.plot(alpha, reg_rmse_train, '-o', color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Test RMSE', color=color)  # we already handled the x-label with ax1
ax2.plot(alpha, reg_rmse_test, '-+', color=color)
ax2.tick_params(axis='y', labelcolor=color)
plt.axhline(y = rmse, color = 'r', alpha=0.6, linestyle = '--', label = "unregularized RMSE")

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.xscale('log')
plt.legend()
plt.title('RMSE vs. Regularization Strength')
plt.show()

#Plot training and testing R2 as a function of regularization strength


fig, ax1 = plt.subplots()

color = 'tab:orange'
ax1.set_xlabel('Regularization Strength [lambda]')
ax1.set_ylabel('Train R2', color=color)
ax1.plot(alpha, reg_r2_train, '-o', color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Test R2', color=color)  # we already handled the x-label with ax1
ax2.plot(alpha, reg_r2_test, '-+', color=color)
ax2.tick_params(axis='y', labelcolor=color)
plt.axhline(y = r2, color = 'r', alpha=0.6, linestyle = '--', label = "unregularized R2")

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.xscale('log')
plt.legend()
plt.title('R2 vs. Regularization Strength')
plt.show()
```

```
lambda_optimal = np.array(alpha)[reg_rmse_test == min(reg_rmse_test)]
r2_optimal = np.array(reg_r2_test)[alpha == lambda_optimal]
rmse_optimal = np.array(reg_rmse_test)[alpha == lambda_optimal]
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Dec  5 09:40:23 2023

@author: tanvibansal
Capstone Q7

"""

import pandas as pd
import numpy as np
import random
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import svm
from sklearn.metrics import RocCurveDisplay,roc_auc_score,accuracy_score,classification_report

random.seed(13839901)

df_raw = pd.read_csv('spotify52kData.csv')
df = df_raw.drop_duplicates(subset=['artists','track_name', 'mode', 'valence'])
#ratings = pd.read_csv('starRatings.csv')

mode = df['mode']
valence = df['valence']

#plot predictor space
plt.hist(valence[mode == 0],alpha=0.5, label='Minor')
plt.hist(valence[mode == 1],alpha=0.5, label='Major')
plt.legend()
plt.xlabel('Valence')
plt.ylabel('Frequency')
plt.title('Valence Histogram Grouped by Mode')

def logistic_regression_finder(x_train, x_test, y_train, y_test, feature):
    from sklearn import metrics
    #plot input predictor space to visualize
    plt.figure(figsize=(15,5))
    plt.suptitle("%s"%(feature),fontsize = 'xx-large')

    #find the widest margin classifier (for the movie of interest) between the average user
ratings and the labelled outcomes
    #select data vectors of interest and transform to numpy arrays of dimension nx1
    #feed these as inputs to support vector classifier model

    train_sort = np.argsort(x_train,axis=0)
    x_train_sorted = np.take_along_axis(x_train, train_sort, axis=0)
    y_train_sorted = y_train[train_sort].ravel()

    test_sort = np.argsort(x_test,axis=0)
    x_test_sorted = np.take_along_axis(x_test, test_sort, axis=0)
    y_test_sorted = y_test[test_sort].ravel()

    x_train_sorted = x_train_sorted.reshape(-1,1)
    x_test_sorted = x_test_sorted.reshape(-1,1)

    #y_train_sorted = y_train_sorted.reshape(-1,1)
    #y_test_sorted = y_test_sorted.reshape(-1,1)


    #fit the model to the training set and use the model to predict the outcomes of the test
set
    model = LogisticRegression(penalty= 'l2', solver='liblinear',C =
100000.0,random_state=13839901).fit(x_train_sorted, y_train_sorted)
    p = model.predict_proba(x_test_sorted)
```

```
    y_pred = model.predict(x_test_sorted)
    inflection_point = x_test_sorted[p[:,1] >= .5][0][0]

    #plot the test set results vs. the raw test set data
    #plt.figure()
    plt.subplot(1,2,1)
    plt.scatter(x_test_sorted,y_test_sorted,alpha=.7)
    plt.plot(x_test_sorted, p[:,1],color='black')
    plt.axhline(y = 0.5, color='black', alpha=0.6, linestyle = 'dotted')
    plt.plot([inflection_point,inflection_point],[0,1],"k--",lw=1)
    plt.xlabel("%s"%(feature))
    plt.ylabel("p(y=1)")
    plt.title("Logistic Regression Fit and Test Sample Data")
    plt.legend(['test sample data','estimated logistic regression line','confusion matrix
boundaries'])

    fpr, tpr, _ = metrics.roc_curve(y_test_sorted,  p[:,1])
    auc = metrics.roc_auc_score(y_test_sorted, p[:,1])

    #create ROC curve
    #plt.figure()
    plt.subplot(1,2,2)
    plt.plot(fpr,tpr)
    plt.plot(np.linspace(0,1,11),np.linspace(0,1,11),color="grey",alpha=.5,linestyle='dashed')
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.title("ROC Curve of Model for Test Set")
    plt.legend(['ROC','Unity'])
    plt.show()

    #estimate betas
    beta1 = model.coef_
    beta0 = model.intercept_

    metrics = {"Predictor":feature,
"AUC":auc,"Beta0":beta0,"Beta1":beta1,"Inflection":inflection_point}
    return metrics

#drop duplicates for all features of interest and split the train/test set to be used for each
model
features = ['popularity', 'key', 'time_signature', 'duration', 'danceability', 'energy',
'loudness', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', 'valence' ,
'tempo']

duplicate_subset_to_drop = ['artists','track_name'] + features
df_f = df_raw.drop_duplicates(subset=duplicate_subset_to_drop)

x = df_f.drop(columns='mode')
y=df_f['mode']

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,random_state =
13839901)



#run the 1-dimension logistic regression for each feature to use as predictors for mode
#collect the metrics for the logistic regression in a list of dictionaries and convert to
dataframe
song_feature_model_metrics = []
for f in features:
    x_train_f = x_train.loc[:,f].values
    x_test_f = x_test.loc[:,f].values
    y_train_f = y_train.values
    y_test_f = y_test.values

    metrics = logistic_regression_finder(x_train_f, x_test_f, y_train_f, y_test_f, f)
    song_feature_model_metrics.append(metrics)
```

```
song_feature_model_metrics = pd.DataFrame(song_feature_model_metrics)


#run multi-dimensional linear SVM classification model using all features of interest in the
above list
x_train_a = x_train.loc[:,features]
y_train_a = np.array(y_train)

x_test_a = x_test.loc[:,features]
y_test_a = np.array(y_test)

clf = svm.SVC()
clf.fit(x_train_a, y_train_a)
y_pred = clf.predict(x_test_a)

svc_disp = RocCurveDisplay.from_estimator(clf, x_test_a, y_test_a)
plt.show()

svm_auc_roc = roc_auc_score(y_test_a, y_pred)


#now try running multi-dimensional non-linear classifiers using all features in above list  as
predictors
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

# Decision Tree
tree_model = DecisionTreeClassifier(random_state=13839901).fit(x_train_a, y_train_a)
tree_predictions = tree_model.predict(x_test_a)
# Output some metrics
tree_auc_roc = roc_auc_score(y_test_a, tree_predictions)
tree_classification_report = classification_report(y_test_a, tree_predictions)
print(tree_classification_report)

# Random Forest
forest_model = RandomForestClassifier(random_state=3839901).fit(x_train_a, y_train_a)
forest_predictions = forest_model.predict(x_test_a)
# Output some metrics
forest_auc_roc = roc_auc_score(y_test_a, forest_predictions)
forest_classification_report = classification_report(y_test_a, forest_predictions)
print(forest_classification_report)
```

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Dec  5 12:35:48 2023

@author: tanvibansal
Capstone Q10
"""

import pandas as pd
import numpy as np
import random
from scipy import spatial
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)
random.seed(13839901)


df = pd.read_csv('spotify52kData.csv')
ratings = pd.read_csv('starRatings.csv')

#pre-processing stage; label song numbers in ratings df, remove songs with fewer than 5 ratings, center each
user's ratings
ratings.columns = range(0,5000)

e = np.sum(~np.isnan(ratings),axis=0)
ind_to_drop = np.where(np.any(e<5))

avg_usr_rating = np.mean(ratings,axis=1)
ratings_centered = np.array(ratings) - np.array(avg_usr_rating).reshape(-1,1)

precision_list = []
recall_list = []
mixtape_song_list = []

for i in range(0,9999):
    #function to find list of similar users to the user of interest
    #find distances between user i and all other users
    pairwise_dist = []

    for j in range(9999):
        u = ratings_centered[i, :]
        v = ratings_centered[j, :]

        ind_to_keep = np.where((~np.isnan(u)) & (~np.isnan(v)))

        dist = spatial.distance.euclidean(u[ind_to_keep], v[ind_to_keep])

        pairwise_dist.append({"Subject": i, "User":j, "Distance": dist})

    pairwise_dist = pd.DataFrame(pairwise_dist)

    #find the top 10% of nearest user neighbors to the subject excluding distance = 0
    ppf = np.percentile(pairwise_dist['Distance'],1)
    similar_users = pairwise_dist[['User','Distance']].loc[np.where((pairwise_dist['Distance'] <= ppf) &
(pairwise_dist['Distance'] != 0))]


    #function to find the predicted ratings a user would give to a certain item based on
    #the weighted average of the ratings from similar users
    songNumber = random.sample(range(0,5000), 1)[0]
    neighbors = similar_users.iloc[:,0]
    distance = similar_users.iloc[:,1]

    similarity = distance**-1
    similar_user_ratings = ratings_centered[neighbors,:].T
    similar_user_ratings_filled = np.nan_to_num(similar_user_ratings,nan=0)

    weighted_sum = np.matmul(similar_user_ratings_filled,similarity)

    weight_inds = np.nan_to_num(similar_user_ratings,nan=0)
    weight_inds[np.where(weight_inds != 0)] = 1
    sims = np.matmul(weight_inds, similarity)**-1

    weight_avg = weighted_sum*sims
```

```
    #find the selected mixtape for each user

    #filter off songs we do not have explicit feedback for. loss of independent data here is compensated for
by increasing the
    #percentage of similar users included in the predicted rating estimation

    songs_to_pred=range(5000)
    #songs_to_pred = np.where(~np.isnan(ratings_centered[i,:]))[0]
    pred_ratings = weight_avg[songs_to_pred]

    usr_i_pred_ratings =
pd.DataFrame({"SongNumber":songs_to_pred,"PredRating":pred_ratings}).sort_values('PredRating',ascending=False)

    mixtape_selection_i = usr_i_pred_ratings.iloc[0:10,:]
    mixtape_song_list.append({"User": i, "Mixtape": mixtape_selection_i['SongNumber'].values})

    if i%100 == 0:
        print("%s users complete"%(i))

mixtape_song_list_ = pd.DataFrame(mixtape_song_list)

#compute precision and recall for all 10k mixtapes
num_ratings = []
precision_list = []
recall_list = []
for i in range(9999):
    u = mixtape_song_list_.loc[i,'Mixtape']['User']
    ratings_c_u = ratings_centered[u,:]
    r_c_u_f = ratings_c_u[np.where(~np.isnan(ratings_c_u))]

    thresh_u = np.percentile(r_c_u_f,75)
    relevant_u = np.where((ratings_c_u >= thresh_u) & (~np.isnan(ratings_c_u)))[0]

    recommended_u = mixtape_song_list_.loc[i,'Mixtape']['Mixtape']
    missing_u = ratings_c_u[recommended_u]

    recommended_u_clean = recommended_u[~np.isnan(missing_u)]

    if len(recommended_u_clean) == 0:
       # precision_u = None
        #recall_u = None
        num_ratings.append(0)
    else:
        precision_u = len([r for r in recommended_u_clean if r in relevant_u])/len(recommended_u_clean)
        recall_u = len([r for r in recommended_u_clean if r in relevant_u])/len(relevant_u)
        num_ratings.append(len(recommended_u_clean))

    precision_list.append(precision_u)
    recall_list.append(recall_u)

np.mean(precision_list)
np.mean(recall_list)
np.mean(num_ratings)



#aggregate metrics over all 10k mixtapes
mixtape_song_list_ = pd.DataFrame(mixtape_song_list)
recommended_songs  = []

for k in range(len(mixtape_song_list)):
    recommended_songs = recommended_songs + list(mixtape_song_list[k]['Mixtape'])
song_counts = pd.Series(recommended_songs).value_counts().sort_values(ascending=False)
most_recommended = df[['track_name','artists']].loc[song_counts[0:15].index.values]

results = pd.DataFrame({"Precision": precision_list, "Recall": recall_list, "Mixtape": mixtape_song_list})
results.to_csv(r"RecommenderSystemResults_NoNew.csv")
```