CS109A Introduction to Data Science: Spotify Final Project

Harvard University
Fall 2018
Instructors: Pavlos Protopapas, Kevin Rader
Group Number: 49
Group Members: Tejal Patwardhan, Akshitha Ramachandran, Grace Zhang

---

In [43]:
```
#RUN THIS CELL
import requests
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/20
18-CS109A/master/content/styles/cs109.css").text
HTML(styles)
```

Out[43]:

In [44]:
```
# import necessary notebooks
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt

import statsmodels.api as sm
from statsmodels.api import OLS

from sklearn import preprocessing
from sklearn.utils import resample
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split, KFold
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import LassoCV
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.preprocessing import PolynomialFeatures
from pandas.plotting import scatter_matrix
import seaborn as sns
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import StratifiedKFold

sns.set(style='whitegrid')
pd.set_option('display.width', 1500)
pd.set_option('display.max_columns', 100)

import random

%matplotlib inline
```

Data Collection and Cleaning

We collected our data by using the Spotify API to create a .csv file of tracks and their features. Grace manually created 2 separate playlists, where one playlist includes random songs that Grace would include in her playlist and the other playlist includes random songs that Grace would not include in her playlist. We used the Spotify API `user_playlist_tracks` endpoint to collect some features, including `track_id`s, of the tracks in each of these playlists. We then used the `audio_features` endpoint of the Spotify API to get additional features like `danceability` for each of our tracks. Finally, we added the `in_playlist` feature to each of our tracks and wrote our final object to `spotify.csv`.

Data Description

Our data includes the following features:

- `danceability`: Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.
- `energy`: Energy represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy. A value of 0.0 is least energetic and 1.0 is most energetic.
- `key`: The estimated overall key of the track. Integers map to pitches using standard Pitch Class Notation. For example, 0 = C, 1 = C♯/D♭, 2 = D, and so on. If no key was detected, the value is -1.
- `loudness`: The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values range between -60 and 0 db.
- `mode`: Mode represents the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Mode is a binary variable; major is represented by 1 and minor is 0.
- `speechiness`: Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.
- `acousticness`: A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.
- `instrumentalness`: Predicts whether a track contains no vocals. “Ooh” and “aah” sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly “vocal”. The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0.
- `liveness`: Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.
- `valence`: A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).
- `tempo`: The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.
- `duration_ms`: The duration of the track in milliseconds.
- `time_signature`: An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure).

- `popularity`: The popularity of a track is a value between 0 and 100, with 100 being the most popular. The popularity is calculated by algorithm and is based, in the most part, on the total number of plays the track has had and how recent those plays are.Generally speaking, songs that are being played a lot now will have a higher popularity than songs that were played a lot in the past.
- `in_playlist`: Response variable. Categorical variable for whether in playlist of desire. 1 if in playlist, 0 if not in playlist.

The following features were recorded to help with visualization later, but not used as predictors in our analysis, as they are not characteristics of the music itself.

- `name`: Song title.
- `artist`: First artist of song.
- `type`: The object type, always deemed "audio_features."
- `id`: The Spotify ID for the track.
- `uri`: The Spotify URI for the track.
- `track_href`: A link to the Web API endpoint providing full details of the track.
- `analysis_url`: An HTTP URL to access the full audio analysis of this track. An access token is required to access this data

Exploratory Data Analysis

In [45]:
```
random.seed(1)
```

In [46]:
```
# load in dataset
spotify_df = pd.read_csv("data/spotify-more2.csv")

# drop unnecessary columns
spotify_df = spotify_df.drop(columns=['type', 'id', 'uri', 'track_href',
 'analysis_url', 'name', 'artist', 'Unnamed: 0'])
```

In [47]:
```
# display head of data
display(spotify_df.head())
```

	acousticness	danceability	duration_ms	energy	in_playlist	instrumentalness	key	liv
0	0.929	0.516	138760	0.0663	0	0.000972	7	0.1
1	0.539	0.454	324133	0.2600	0	0.000780	8	0.0
2	0.360	0.676	205773	0.4400	0	0.000069	0	0.1
3	0.984	0.466	294307	0.0718	0	0.000931	0	0.1
4	0.779	0.496	423573	0.6340	0	0.402000	5	0.0

In [48]:
```
# display shape of data
display(spotify_df[spotify_df["in_playlist"]==0].shape)
```

```
(2500, 15)
```

We have 5060 songs in our initial analysis. 2650 are included in Grace's playlist, and 2500 are not included in Grace's playlist.

In [49]:
```
# generate summary chart of features
features = []
means = []
var = []
ranges = []
mins = []
maxes = []

for feature in spotify_df:
    if feature != "in_playlist":
        features.append(feature)
        means.append(spotify_df[feature].mean())
        var.append(spotify_df[feature].var())
        ranges.append(spotify_df[feature].ptp())
        mins.append(spotify_df[feature].min())
        maxes.append(spotify_df[feature].max())

summary_df = pd.DataFrame(data = {'feature': features,
                                  'mean': means,
                                  'var' : var,
                                  'range': ranges,
                                  'min': mins,
                                  'max': maxes})
```

Below are summary statistics for all the features we plan to analyze:

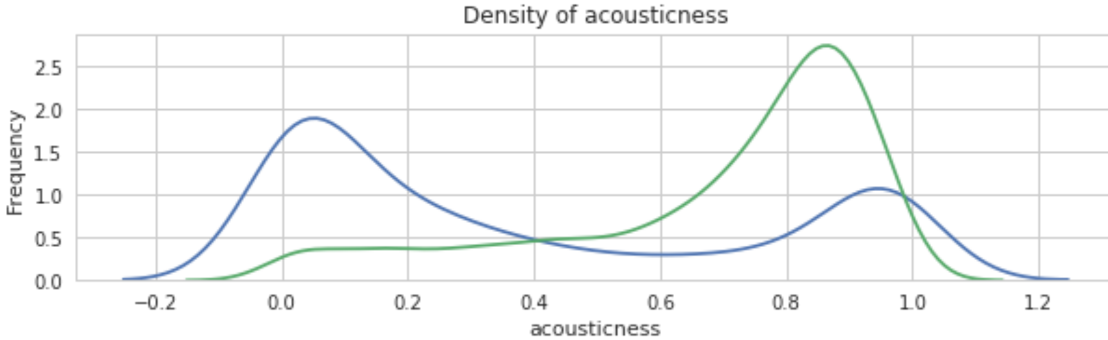
In [50]:
```
display(summary_df)
```

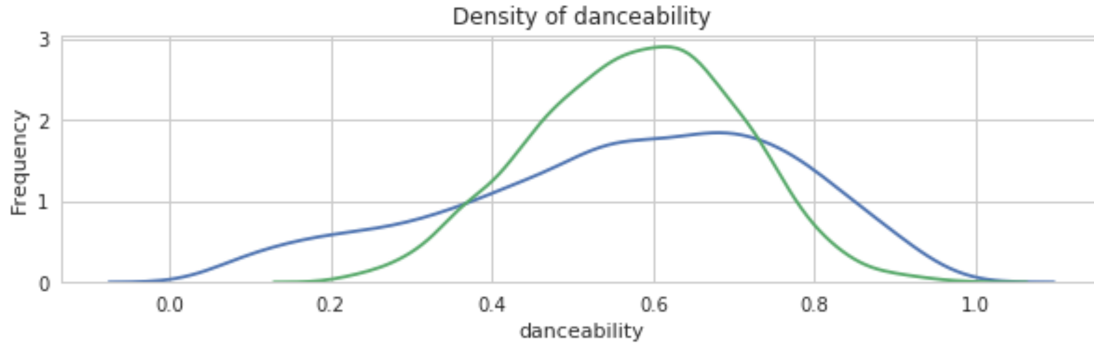
	feature	mean	var	range	min	
0	acousticness	0.540199	1.267884e-01	9.959953e-01	0.000005	0.996
1	danceability	0.570920	2.931912e-02	9.162000e-01	0.061800	0.978
2	duration_ms	245718.492885	1.911563e+10	3.346533e+06	44507.000000	3391040.
3	energy	0.439224	6.633419e-02	9.901450e-01	0.000855	0.991
4	instrumentalness	0.143138	9.302492e-02	9.870000e-01	0.000000	0.987
5	key	5.223913	1.251578e+01	1.100000e+01	0.000000	11.000
6	liveness	0.163377	1.798945e-02	9.800000e-01	0.012000	0.992
7	loudness	-10.270219	3.464989e+01	4.217600e+01	-42.476000	-0.300
8	mode	0.650198	2.274856e-01	1.000000e+00	0.000000	1.000
9	popularity	36.977470	4.773025e+02	1.000000e+02	0.000000	100.000
10	speechiness	0.070655	6.217856e-03	8.989000e-01	0.023100	0.922
11	tempo	117.657563	8.604272e+02	1.790410e+02	42.581000	221.622
12	time_signature	3.919763	1.655315e-01	5.000000e+00	0.000000	5.000
13	valence	0.425801	5.455384e-02	9.591000e-01	0.025900	0.985

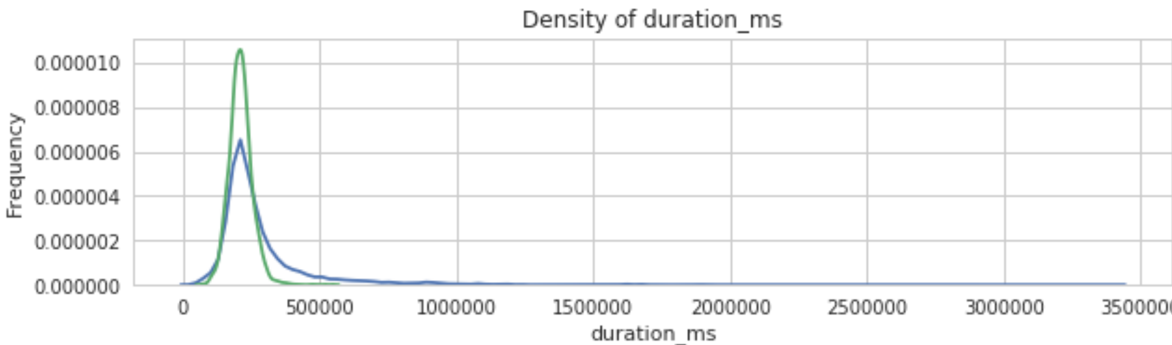
We can see that all features have values that are expected as per the Spotify API documentation. To analyze each feature in more granularity we looked at density plots.

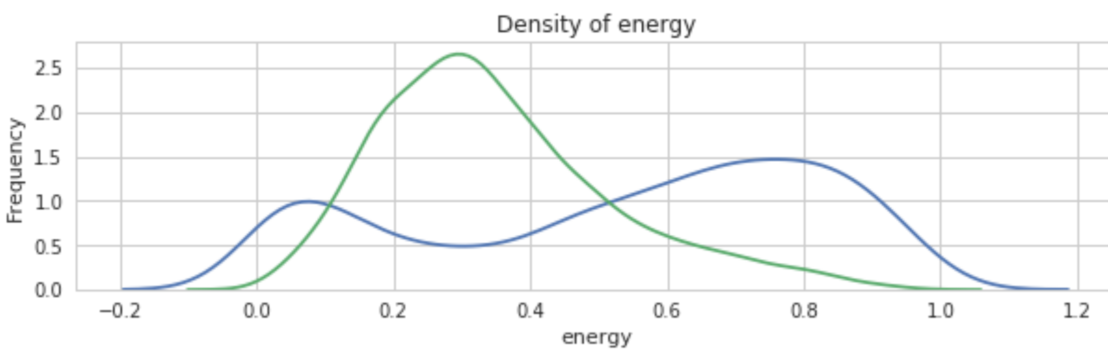
In [51]:
```
# define response column
response_col = 'in_playlist'
```

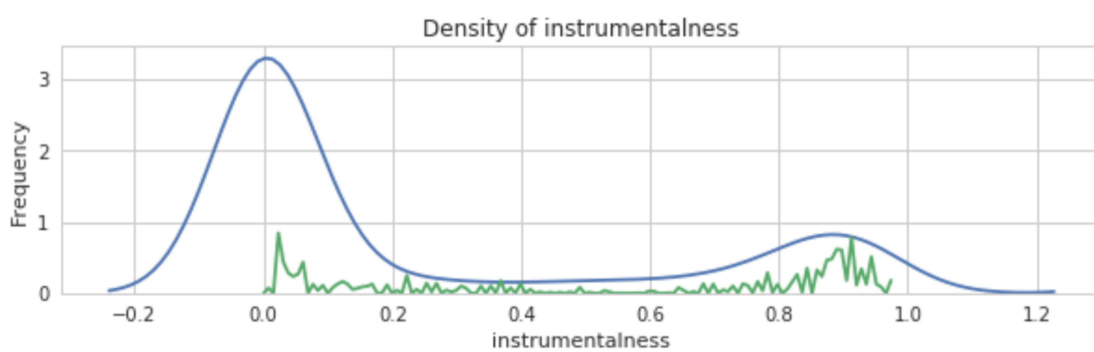
In [52]:
```
# prepare data for display
resp_col_loc = list(spotify_df.columns).index('in_playlist')
spotify_graphs_df = spotify_df.drop(columns=[response_col])
num_cols = len(spotify_graphs_df.columns)
nbin = 15

# iterate through all the features and display them
fig, axs = plt.subplots(num_cols, 1, figsize=(10,50))
for i in range(num_cols):
    sns.distplot(spotify_graphs_df[spotify_df.in_playlist == 0][spotify_
graphs_df.columns[i]], hist = False, kde = True, ax=axs[i])
    sns.distplot(spotify_graphs_df[spotify_df.in_playlist == 1][spotify_
graphs_df.columns[i]], hist = False, kde = True, ax=axs[i])
    axs[i].set_title("Density of " + str(spotify_graphs_df.columns[i]))
    axs[i].set_ylabel(r'Frequency')
fig.subplots_adjust(hspace=.5)
plt.show()
```


Density of acousticness
Frequency
acousticness

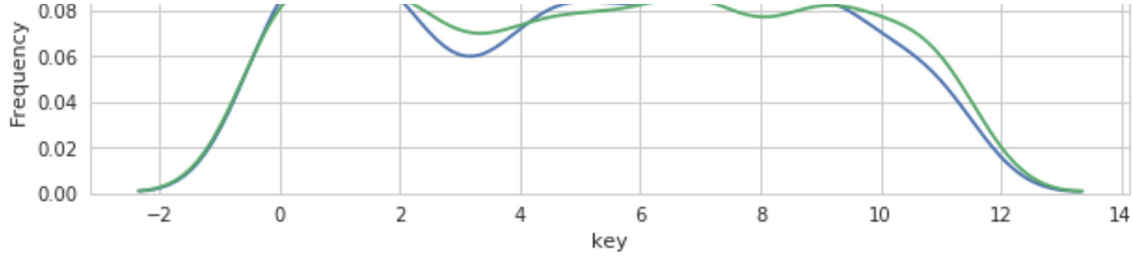

Density of danceability
Frequency
danceability

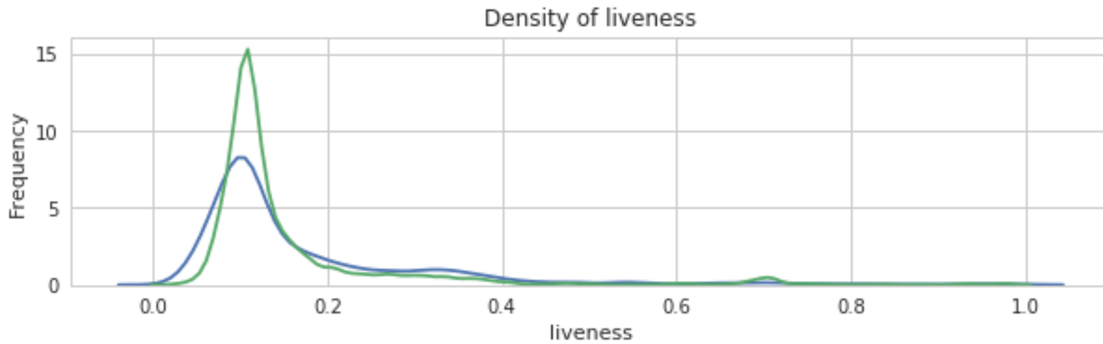

Density of duration_ms
Frequency
duration_ms

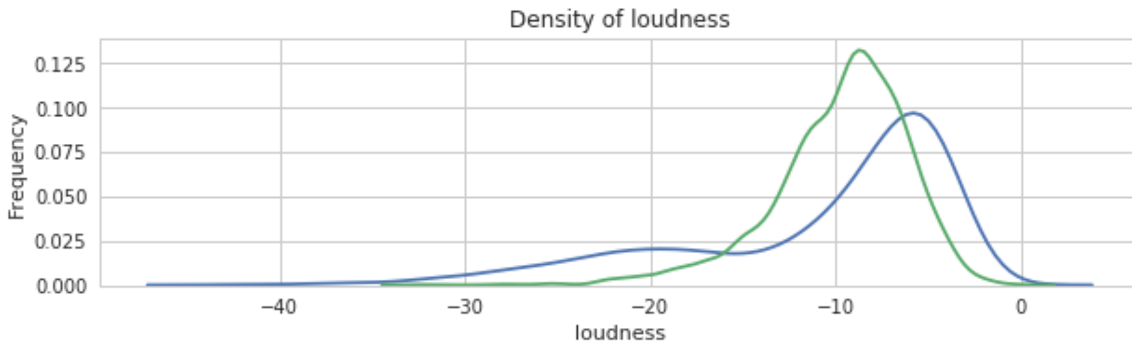

Density of energy
Frequency
energy

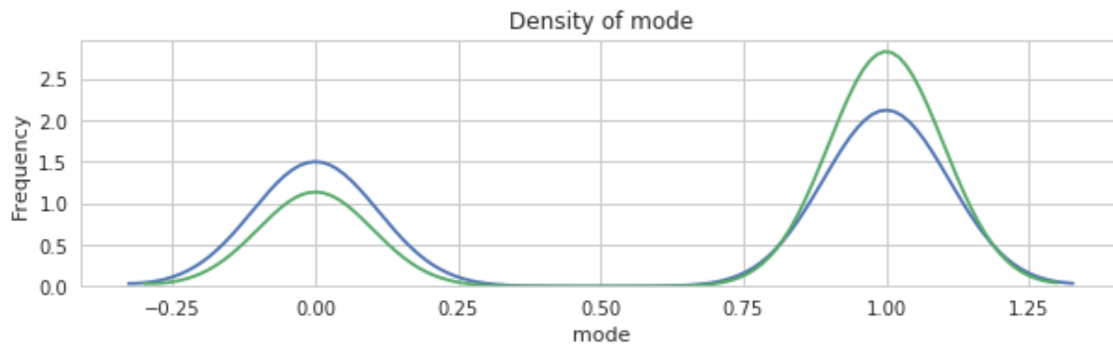

Density of instrumentalness
Frequency
instrumentalness

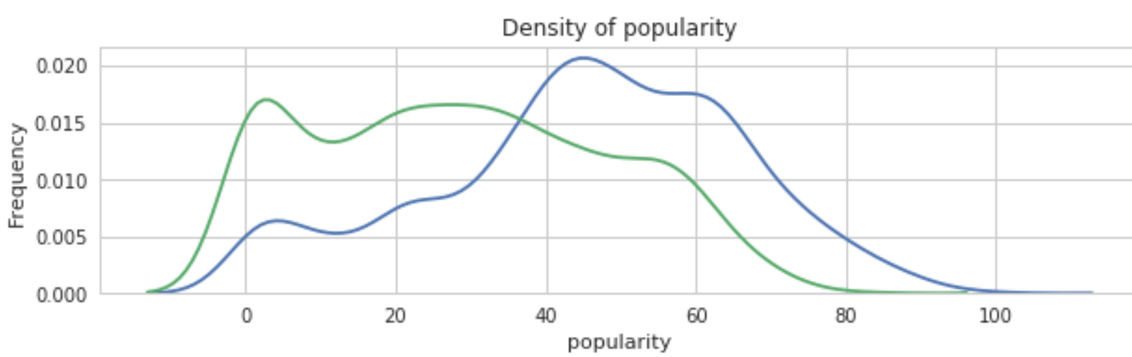

Density of key
0.10

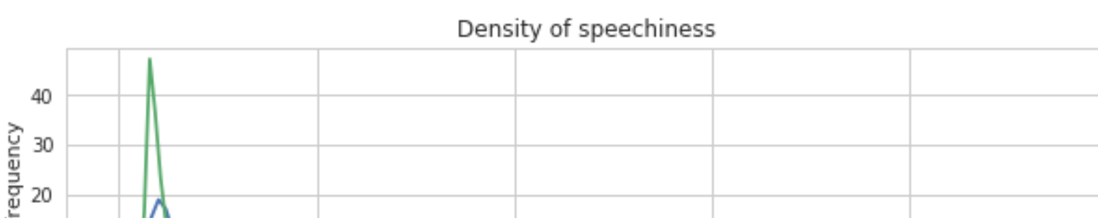

Frequency
0.08
0.06
0.04
0.02
0.00
−2 0 2 4 6 8 10 12 14
key

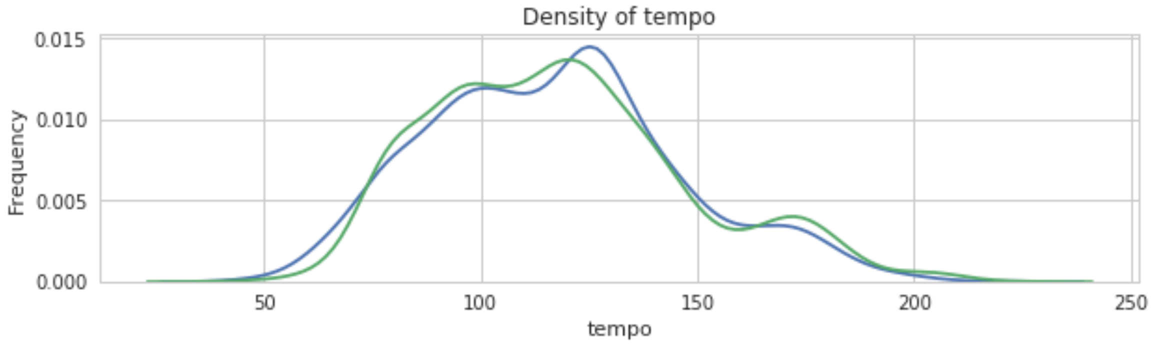

Density of liveness
Frequency
15
10
5
0
0.0 0.2 0.4 0.6 0.8 1.0
liveness

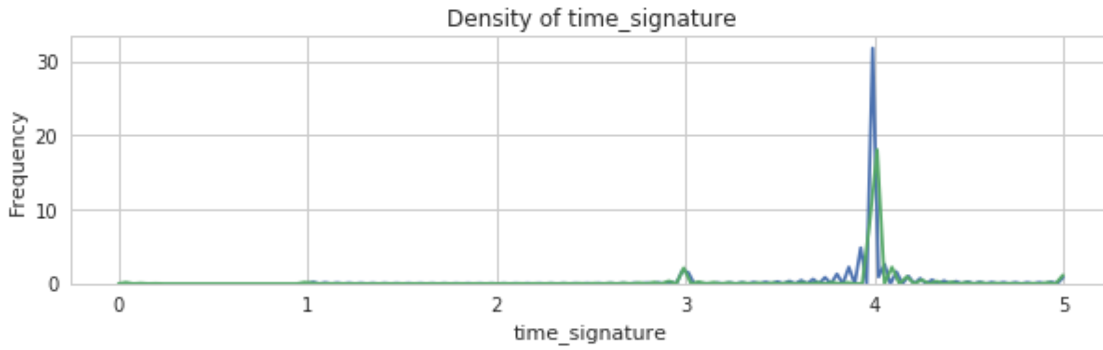

Density of loudness
Frequency
0.125
0.100
0.075
0.050
0.025
0.000
−40 −30 −20 −10 0
loudness

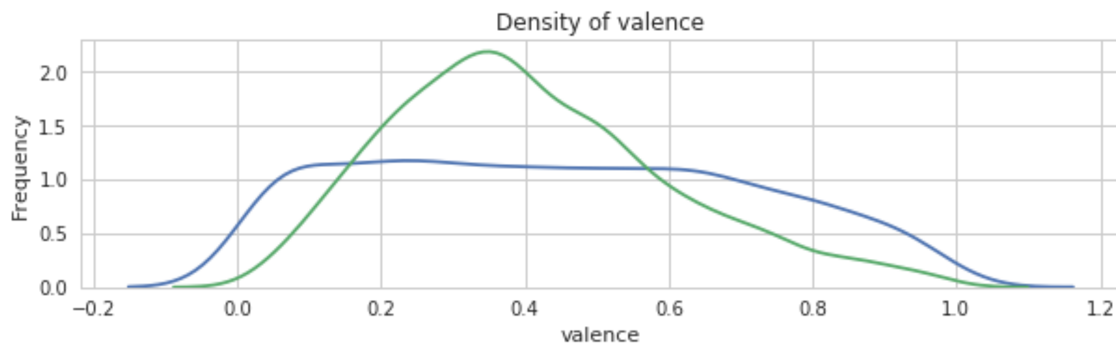

Density of mode
Frequency
2.5
2.0
1.5
1.0
0.5
0.0
−0.25 0.00 0.25 0.50 0.75 1.00 1.25
mode


Density of popularity
Frequency
0.020
0.015
0.010
0.005
0.000
0 20 40 60 80 100
popularity


Density of speechiness
requency
40
30
20



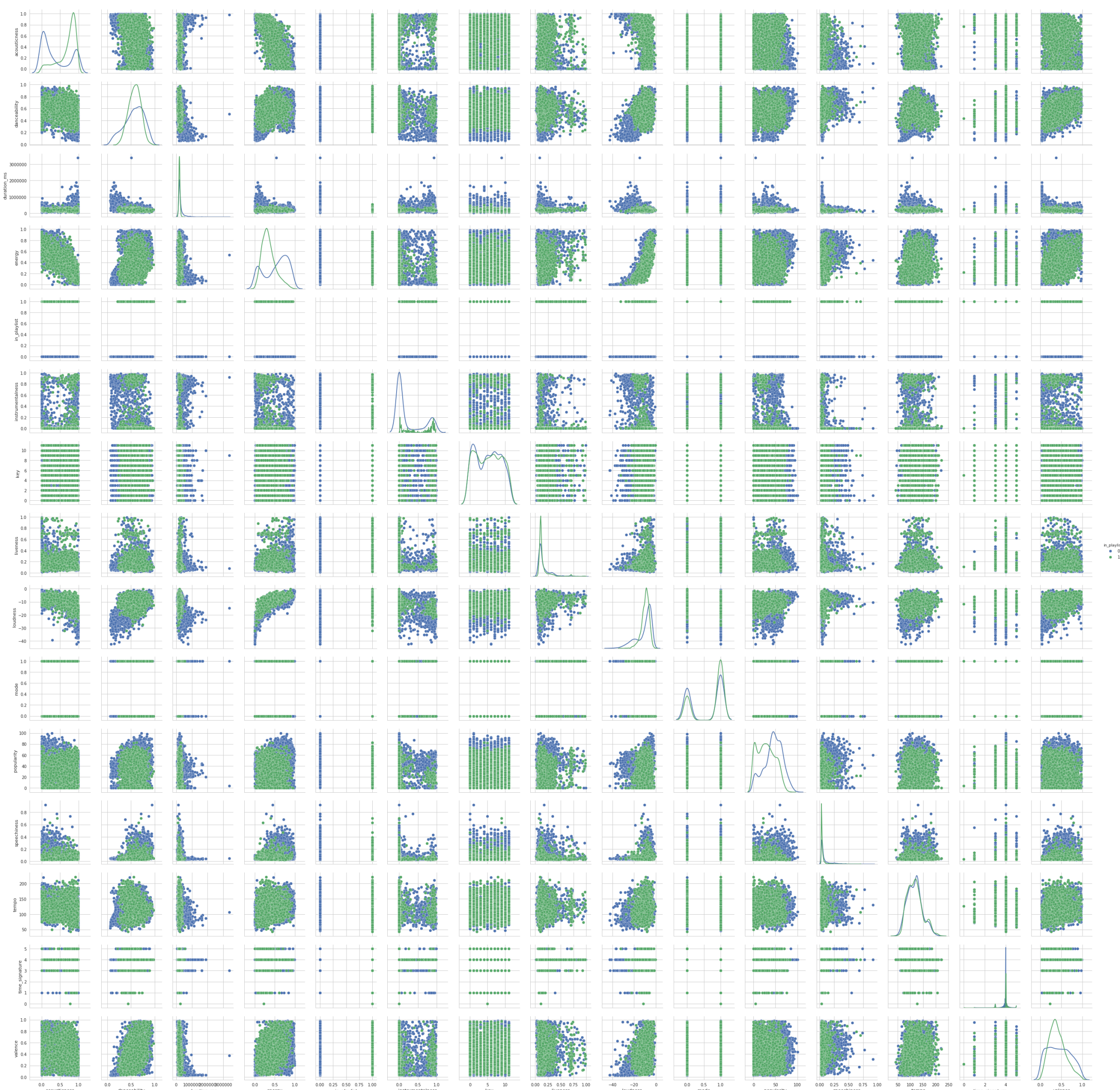




Looking at the density plots above, we note some features that show clear differences in distribution between the playlist and non-playlist. While non-playlist songs contain a roughly uniform distribution of energy values, playlist songs spike at an energy level between 0.2-0.4. Acousticness in playlist tracks is much higher on average, spiking around 0.8, while non-playlist tracks most frequently have acousticness values around 0.1. Instrumentalness is a particularly interesting feature. While the distribution non-playlist tracks is bimodal, peaking at around 0 and 0.9, playlist tracks have a few very well-defined peaks between 0 and 0.3. We will note in advance that this may induce a risk of overfitting based on instrumentalness values. Playlist tracks have lower loudnesses on average, centering around -10, while non-playlist tracks -5. In terms of speechiness, the distribution for playlist tracks has a much lower variance and slightly lower expected value, centering around 0.3 while non-playlist tracks center around 0.4. Valence for non-playlist tracks is roughly uniformly distributed, while playlist tracks demonstrate a roughly normal distribution centered around 0.3. Finally in terms of popularity, playlist tracks show a peak in their distribution around 60, while non-playlist tracks have a more variable distribution with a peak between 45-55. The rest of the features are roughly similar in distribution between playlist and non-playlist tracks.

In [53]:
```
# pair plots
ax = sns.pairplot(spotify_df, hue = "in_playlist", diag_kind="kde")
ax
plt.show()
```

```
/usr/share/anaconda3/lib/python3.6/site-packages/statsmodels/nonparamet
ric/kde.py:488: RuntimeWarning: invalid value encountered in true_divid
e
  binned = fast_linbin(X, a, b, gridsize) / (delta * nobs)
/usr/share/anaconda3/lib/python3.6/site-packages/statsmodels/nonparamet
ric/kdetools.py:34: RuntimeWarning: invalid value encountered in double
_scalars
  FAC1 = 2*(np.pi*bw/RANGE)**2
/usr/share/anaconda3/lib/python3.6/site-packages/numpy/core/_methods.p
y:26: RuntimeWarning: invalid value encountered in reduce
  return umr_maximum(a, axis, None, out, keepdims)
```



The pairplot above demonstrates a few interesting things. First, we notice weakly positive correlations between loudness and energy, loudness and danceability, and danceablility and loudness. We also notice a negative correlation between acousticness and energy. These correlations will be useful to keep in mind if we conduct variable selection or regularization at a later point. Also, none of these pairwise plots show clear separability between the two playlists.

Baseline Logistic Classifier

In [54]:
```
# set seed
random.seed(1)

# split into train and test
train, test = train_test_split(spotify_df, test_size = 0.2, random_state
=50)
x_train, y_train = train.drop(columns=[response_col]), train[response_co
l].values
x_test, y_test = test.drop(columns=[response_col]), test[response_col].v
alues

# create logistic model
log_reg_model = LogisticRegression(C=100000, fit_intercept=False)
log_reg_model.fit(x_train, y_train)

# predict
log_reg_train_predictions = log_reg_model.predict(x_train)
log_reg_test_predictions = log_reg_model.predict(x_test)

# calculate scores
log_reg_train_score = accuracy_score(y_train, log_reg_train_predictions)
log_reg_test_score = accuracy_score(y_test, log_reg_test_predictions)

# display scores
print('[Logistic Regression] Classification accuracy for train set: {}'.
format(log_reg_train_score))
print('[Logistic Regression] Classification accuracy for test set: {}'.f
ormat(log_reg_test_score))
```

```
[Logistic Regression] Classification accuracy for train set: 0.69367588
93280632
[Logistic Regression] Classification accuracy for test set: 0.670948616
6007905
```

Our baseline logistic model is able to achieve an accuracy of roughly 69.4% in the training set, and 67.1% in the test set.

Logistic Classifier with Quadratic Terms

In [55]:
```
# add quadratic terms
x_train_q = x_train.copy()
x_test_q = x_test.copy()

# add quadratic terms
for col in x_train:
    if col != "mode": # our only binary variable
        name = col + "^2" # name column as col^2
        x_train_q[name] = np.square(x_train_q[col])
        x_test_q[name] = np.square(x_test_q[col])

# create logistic model
log_reg_model_q = LogisticRegression(C=100000, fit_intercept=False)
log_reg_model_q.fit(x_train_q, y_train)

# predict
log_reg_train_q_predictions = log_reg_model_q.predict(x_train_q)
log_reg_test_q_predictions = log_reg_model_q.predict(x_test_q)

# calculate scores
log_reg_train_q_score = accuracy_score(y_train, log_reg_train_q_predicti
ons)
log_reg_test_q_score = accuracy_score(y_test, log_reg_test_q_predictions
)

# display scores
print('[Logistic Regression With Quadratic Terms] Classification accurac
y for train set: {}'.format(log_reg_train_q_score))
print('[Logistic Regression With Quadratic Terms] Classification accurac
y for test set: {}'.format(log_reg_test_q_score))
```

```
[Logistic Regression With Quadratic Terms] Classification accuracy for
train set: 0.4965415019762846
[Logistic Regression With Quadratic Terms] Classification accuracy for
test set: 0.4841897233201581
```

When trying to add quadratic terms, we see that the model performs worse. The test and training accuracies are both low at roughly 48.4% and 49.7%.

L1 and L2 Regularization

In [56]:
```
alphas = (.1,.5,1,5,10,50,100)

# L1 regularization
lr_l1_model = LogisticRegressionCV(cv=5, penalty='l1', solver='liblinea
r', max_iter=100000).fit(x_train, y_train)
lr_l2_model = LogisticRegressionCV(cv=5, max_iter=100000).fit(x_train, y
_train)
```

In [57]:
```
def get_lr_cv(model, model_name, x_train=x_train, y_train=y_train, x_tes
t=x_test, y_test=y_test):
    train_predictions = model.predict(x_train)
    train_score = accuracy_score(y_train, train_predictions)
    test_predictions = model.predict(x_test)
    test_score = accuracy_score(y_test, test_predictions)
    test_confusion_matrix = confusion_matrix(y_test, test_predictions)
    print('[{}] Classification accuracy for train set: {}'.format(model_
name, train_score))
    print('[{}] Classification accuracy for test set: {}'.format(model_n
ame, test_score))
    return train_score, test_score, test_confusion_matrix
l1_stats = get_lr_cv(lr_l1_model, 'L1 Reg')
l2_stats = get_lr_cv(lr_l2_model, 'L2 Reg')
```

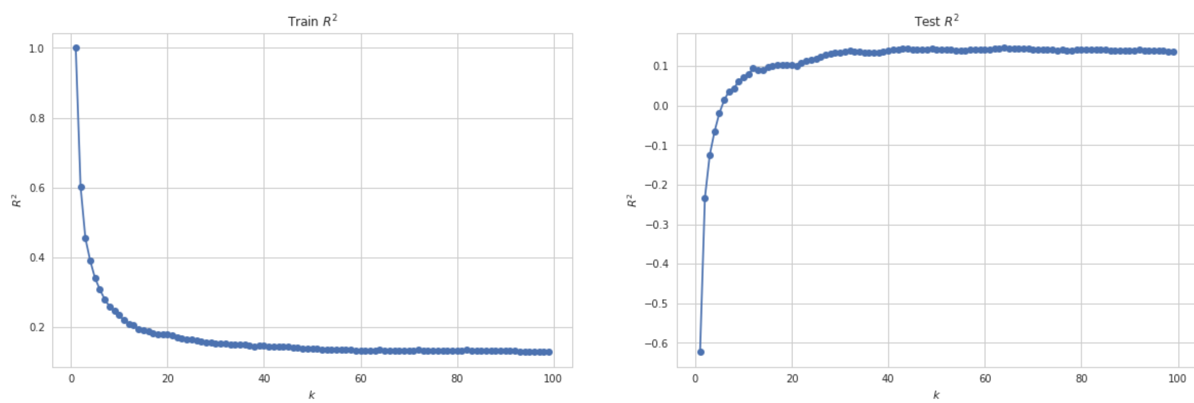
```
[L1 Reg] Classification accuracy for train set: 0.8866106719367589
[L1 Reg] Classification accuracy for test set: 0.8873517786561265
[L2 Reg] Classification accuracy for train set: 0.6926877470355731
[L2 Reg] Classification accuracy for test set: 0.6699604743083004
```

L1 regularization performs much better than L2. The L1 regularized model achieves about 88.8% accuracy in the training data and about 88.9% in the test, well outperforming our baseline model. The L2 regularized model performs on par with our baseline, achieving a training accuracy of around 69.2% and a test accuracy of 66.9%.

kNN

In [58]:
```
# make kNN preds binary
def parsenKKRes(predictions):
    for i in range(len(predictions)):
        if predictions[i] < 0.5:
            predictions[i] = 0
        else:
            predictions[i] = 1
    return predictions
```

In [59]:
```
# make regressor
ks = range(1, 100) # Grid of k's
scores_train = [] # R2 scores
scores_test = [] # R2 scores
acc_train = []
acc_test = []
for k in ks:
    knnreg = KNeighborsRegressor(n_neighbors=k) # Create KNN model
    knnreg.fit(x_train, y_train) # Fit the model to training data
    scores_train.append(knnreg.score(x_train, y_train))  # Calculate R^2
 score
    scores_test.append(knnreg.score(x_test, y_test)) # Calculate R^2 sco
re
    predicted_train = knnreg.predict(x_train)
    predicted_test = knnreg.predict(x_test)
    acc_train.append(accuracy_score(y_train, parsenKKRes(predicted_train
)))
    acc_test.append(accuracy_score(y_test, parsenKKRes(predicted_test)))

# Plot
fig, ax = plt.subplots(1,2, figsize=(20,6))
ax[0].plot(ks, scores_train,'o-')
ax[0].set_xlabel(r'$k$')
ax[0].set_ylabel(r'$R^{2}$')
ax[0].set_title(r'Train $R^{2}$')
ax[1].plot(ks, scores_test,'o-')
ax[1].set_xlabel(r'$k$')
ax[1].set_ylabel(r'$R^{2}$')
ax[1].set_title(r'Test $R^{2}$')
plt.show()
```



In [60]:
```
# determine which k index has best test accuracy
k_index = np.argmax(acc_test)
k_index
```

Out[60]: 81

In [61]:
```
print("[kNN] Classification accuracy for training set: ", acc_train[k_in
dex])
print("[kNN] Classification accuracy for test set: ", acc_test[k_index])
```

```
[kNN] Classification accuracy for training set:  0.6314229249011858
[kNN] Classification accuracy for test set:  0.6590909090909091
```

Our kNN regressor performs at the same level as our baseline logistic classifier. The test set is at a 65.9% accuracy while the training is at 63.1%. Additionally, we see that our R^2 score converges to roughly 0.1, which is not great.

LDA and QDA

In [62]:
```
# LDA
lda = LinearDiscriminantAnalysis()
model_lda = lda.fit(x_train, y_train)
acc_lda = model_lda.score(x_train, y_train)
acc_lda_test = model_lda.score(x_test, y_test)

# print accuracy scores
print("[LDA] Classification accuracy for train set :",acc_lda)
print("[LDA] Classification accuracy for test set :",acc_lda_test)


# QDA
qda = QuadraticDiscriminantAnalysis()
model_qda = qda.fit(x_train, y_train)
acc_qda = model_qda.score(x_train, y_train)
acc_qda_test = model_qda.score(x_test, y_test)
print("[QDA] Classification accuracy for train set:",acc_qda)
print("[QDA] Classification accuracy for test set:",acc_qda_test)
```

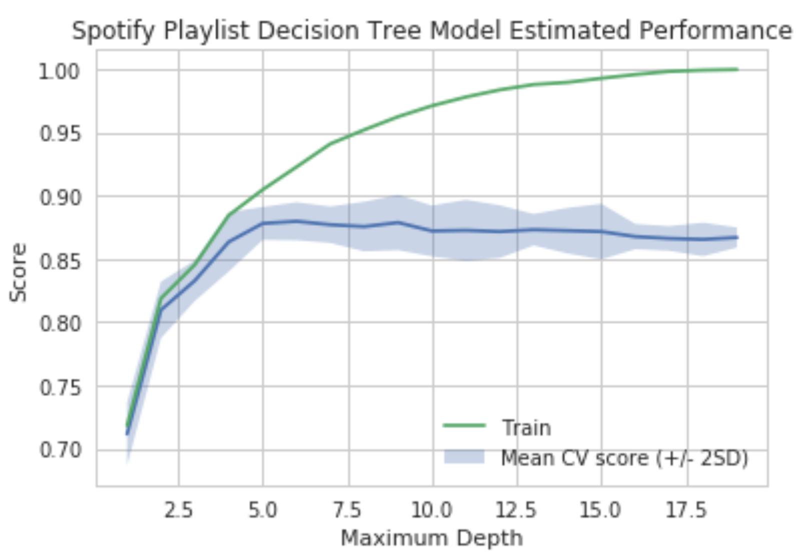
```
[LDA] Classification accuracy for train set : 0.8809288537549407
[LDA] Classification accuracy for test set : 0.8843873517786561
[QDA] Classification accuracy for train set: 0.8656126482213439
[QDA] Classification accuracy for test set: 0.866600790513834
```

LDA performs better than QDA, and both perform above baseline. LDA achieves an accuracy of about 88.1% in the training and 88.4% in the testing data, while QDA ahieves an accuracy of about 86.6% in the training and 86.7% in the testing data.

Decision Trees

In [63]:
```
# classify by depth
def treeClassifierByDepth(depth, x_train, y_train, cvt = 5):
    model = DecisionTreeClassifier(max_depth=depth).fit(x_train, y_train
)
    return cross_val_score(model, x_train, y_train, cv = cvt)
```

In [64]:
```
# 5-fold CV
means = []
lower = []
upper = []
sds = []
trains = []
for i in range(1, 20):
    # fit model
    tc = treeClassifierByDepth(i, x_train, y_train)
    # calc mean and sd
    cur_mean = np.mean(tc)
    cur_sd = np.std(tc)
    train_val = DecisionTreeClassifier(max_depth=i).fit(x_train, y_train
).score(x_train,y_train)
    # add to lists
    trains.append(train_val)
    means.append(cur_mean)
    lower.append(cur_mean - 2*cur_sd)
    upper.append(cur_mean + 2*cur_sd)

plt.plot(range(1,20),means)
plt.fill_between(range(1,20), lower, upper, alpha = 0.3, label = "Mean C
V score (+/- 2SD)")
plt.plot(range(1,20), trains, label="Train")
plt.title("Spotify Playlist Decision Tree Model Estimated Performance")
plt.xlabel("Maximum Depth")
plt.ylabel("Score")
plt.legend()
plt.show()
```



In [65]:
```
# cross validation performance
train_score = means[5]
print("[Decision Tree Classifier] Mean classification accuracy training
 set: ",train_score)
print("Mean +/- 2 SD: (", lower[4],",",upper[4],")")
```

```
[Decision Tree Classifier] Mean classification accuracy training set:
0.8796923499519297
Mean +/- 2 SD: ( 0.8649746226416641 , 0.8909557288057866 )
```

In [66]:
```
# test set performance
model_dec_tree = DecisionTreeClassifier(max_depth=6).fit(x_train, y_trai
n)
test_score = model_dec_tree.score(x_test, y_test)
print("[Decision Tree Classifier] Mean classification accuracy test set:
 ", test_score)
```

```
[Decision Tree Classifier] Mean classification accuracy test set:  0.89
03162055335968
```

We achieve the best cross-validation score at a tree depth of 6, with an accuracy of 88.0%. Additionally, we observe a relatively narrow spread in estimated performances, as there is a roughly 2% difference between +/- two standard deviations. We see that this model also performs quite well in the test set, with an accuracy score of 88.7%, proving superior to all the other models we have tried so far.

Random Forest

In [67]:
```
# config parameters
num_trees = 45
new_depth = 6

# model random forest
model_rf = RandomForestClassifier(n_estimators=num_trees, max_depth=new_
depth)

# fit model on X_train data
model_rf.fit(x_train, y_train)

# predict using model
y_pred_train_rf = model_rf.predict(x_train)
y_pred_test_rf = model_rf.predict(x_test)

# accuracy from train and test
train_score_rf = accuracy_score(y_train, y_pred_train_rf)
test_score_rf = accuracy_score(y_test, y_pred_test_rf)

# print accuracy scores
print("[Random Forest] Classification accuracy for train set: ", train_s
core_rf)
print("[Random Forest] Classification accuracy for test set:", test_scor
e_rf)
```

```
[Random Forest] Classification accuracy for train set:  0.9300889328063
241
[Random Forest] Classification accuracy for test set: 0.922924901185770
8
```

A random forest, at the same depth as the decision tree (namely a depth of 6) performs even better. The test data reaches an accuracy of about 92.6% in the training at 91.5% in the test.

Bagging

Create 45 bootstrapped datasets, fitting a decision tree to each of them and saving their predictions:

In [68]:
```
# bootstrap
bagging_train_arr = []
bagging_test_arr = []
estimators = []

tree_res = []

tree = DecisionTreeClassifier(max_depth=new_depth)

# classify train and test with bootstrap models
for i in range(num_trees):
    boot_x, boot_y = resample(x_train, y_train)
    fit_tree = tree.fit(boot_x, boot_y)
    estimators.append(fit_tree)
    bagging_train_arr.append(tree.predict(x_train))
    bagging_test_arr.append(tree.predict(x_test))
```

Construct dataframes with all the bootstrapped data:

In [69]:
```
# train
bagging_train = pd.DataFrame()
for i in range(len(bagging_train_arr)):
    col_name = "Bootstrap Model " + str(i + 1)
    bagging_train[col_name] = bagging_train_arr[i]

# test
bagging_test = pd.DataFrame()
for i in range(len(bagging_test_arr)):
    col_name = "Bootstrap Model " + str(i + 1)
    bagging_test[col_name] = bagging_test_arr[i]

# generate renaming row obj
rename = {}

for i in range(0, 1104):
    rename[i] = "Training Row " + str(i + 1)


bagging_train.rename(rename, inplace=True)
bagging_test.rename(rename,  inplace=True)
```

Combine predictions from all the bootstraps and assess how the model performs:

In [70]:
```
# combining all data points from the data to determine accuracy
y_preds_train = []
y_preds_test = []

for row in bagging_train.iterrows():
    if np.mean(row[1]) > 0.5:
        y_preds_train.append(1)
    else:
        y_preds_train.append(0)

for row in bagging_test.iterrows():
    if np.mean(row[1]) > 0.5:
        y_preds_test.append(1)
    else:
        y_preds_test.append(0)

def compare_acc(preds, actual):
    count = 0
    for i in range(len(preds)):
        if preds[i] == actual.item(i):
            count += 1
    return(count/len(preds))

bagging_train_score = compare_acc(y_preds_train,y_train)
bagging_test_score = compare_acc(y_preds_test,y_test)

print("[Bagging] Classification accuracy for train set: ", bagging_train
_score)
print("[Bagging] Classification accuracy for test set: ", bagging_test_s
core)
```

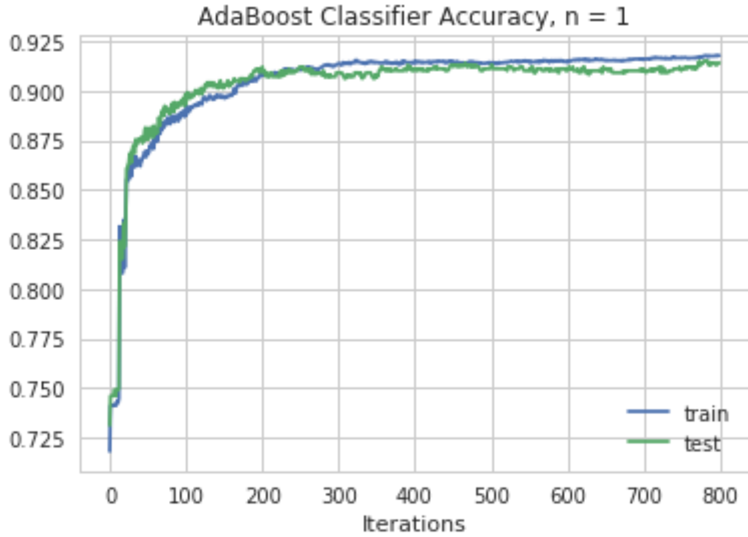
```
[Bagging] Classification accuracy for train set:  0.9370059288537549
[Bagging] Classification accuracy for test set:  0.9150197628458498
```

The model clearly performed better after using bootstrapped data to fit it. It has increased from 88% on the training data to 94.0%, and from 88.1% on the test data to 90.4%. This makes bagging the most accurate model we have tried so far.

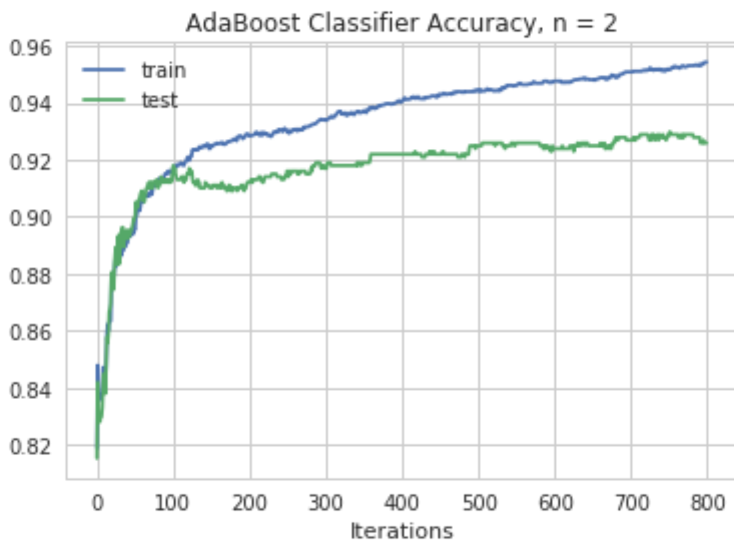
Boosting

In [71]:
```
# define classifier function
def boostingClassifier(x_train, y_train, depth):
    # AdaBoostClassifier
    abc = AdaBoostClassifier(DecisionTreeClassifier(max_depth=depth),
                         n_estimators=800, learning_rate = 0.05)
    abc.fit(x_train, y_train)
    # staged_score train to plot
    abc_predicts_train = list(abc.staged_score(x_train,y_train))
    plt.plot(abc_predicts_train, label = "train");

    # staged_score test to plot
    abc_predicts_test = list(abc.staged_score(x_test,y_test))
    plt.plot(abc_predicts_test, label = "test");

    plt.legend()
    plt.title("AdaBoost Classifier Accuracy, n = "+str(depth))
    plt.xlabel("Iterations")
    plt.show()

    return("Maximum test accuracy for depth of "+str(depth)+" is "+str(m
ax(abc_predicts_test))+" at "+str(abc_predicts_test.index(max(abc_predic
ts_test)))+" iterations")
```

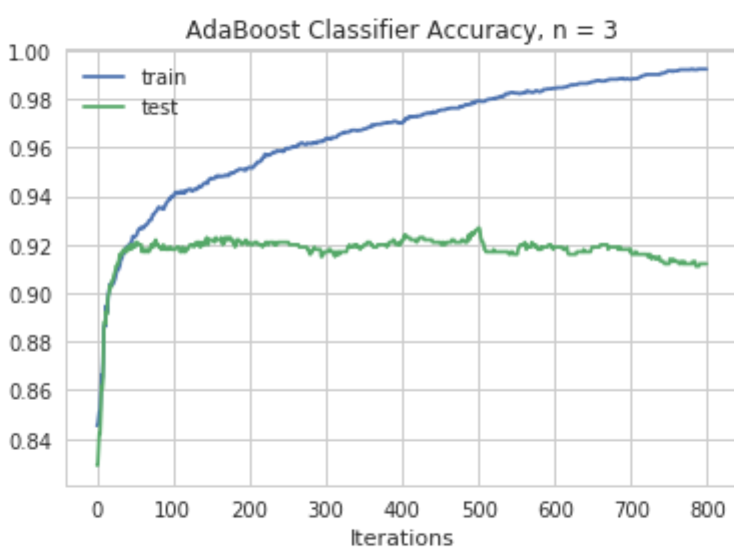
In [72]:
```
for i in range(1,5):
    print(boostingClassifier(x_train, y_train, i))
```



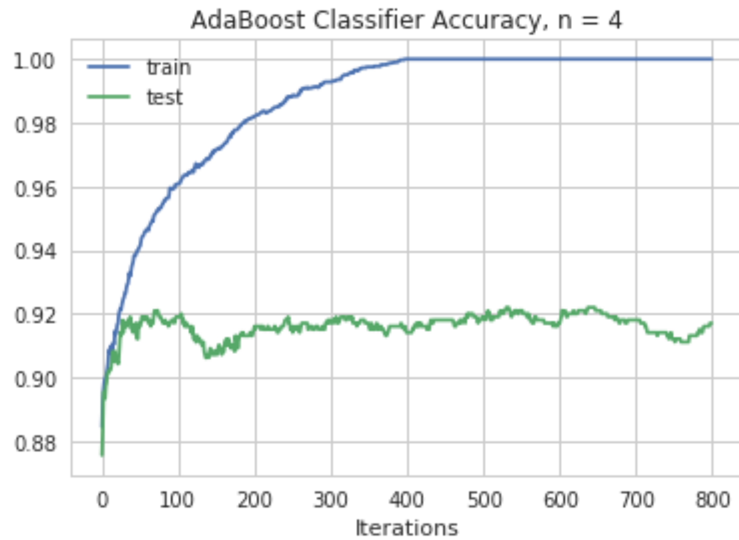

```
Maximum test accuracy for depth of 1 is 0.9150197628458498 at 773 itera
tions
```




```
Maximum test accuracy for depth of 2 is 0.9298418972332015 at 751 itera
tions
```



```
Maximum test accuracy for depth of 3 is 0.9268774703557312 at 500 itera
tions
```




```
Maximum test accuracy for depth of 4 is 0.9219367588932806 at 530 itera
tions
```

We see based upon an AdaBoostClassifier the maximum test accuracy of 93.0% is attained at a depth of 2. This is attained after 751 iterations. The AdaBoostClassifier is our most accurate model so far.

Neural Networks

We next created an artificial neural network to classify our playlist songs.

In [73]:
```
# check input and output dimensions
input_dim_2 = x_train.shape[1]
output_dim_2 = 1
print(input_dim_2,output_dim_2)
```

```
14 1
```

In [74]:
```
# create sequential multi-layer perceptron
model2 = Sequential()

# initial layer
model2.add(Dense(10, input_dim=input_dim_2,
                activation='relu'))

# second layer
model2.add(Dense(10, input_dim=input_dim_2,
                activation='relu'))

# third layer
model2.add(Dense(10, input_dim=input_dim_2,
                activation='relu'))

# output layer
model2.add(Dense(1, activation='sigmoid'))

# compile the model
model2.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['ac
curacy'])
model2.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_46 (Dense)             (None, 10)                150
_________________________________________________________________
dense_47 (Dense)             (None, 10)                110
_________________________________________________________________
dense_48 (Dense)             (None, 10)                110
_________________________________________________________________
dense_49 (Dense)             (None, 1)                 11
=================================================================
Total params: 381
Trainable params: 381
Non-trainable params: 0
_________________________________________________________________
```

In [75]:
```
# fit the model
model2_history = model2.fit(
    x_train, y_train,
    epochs=50, validation_split = 0.5, batch_size = 128, verbose=False)
```

In [76]:
```
# model loss
print("[Neural Net - Model 1] Loss: ", model2_history.history['loss'][-1
])
print("[Neural Net - Model 1] Val Loss: ", model2_history.history['val_l
oss'][-1])
print("[Neural Net - Model 1] Test Loss: ", model2.evaluate(x_test, y_te
st, verbose=False))
print("[Neural Net - Model 1] Accuracy: ", model2_history.history['acc']
[-1])
print("[Neural Net - Model 1] Val Accuracy: ", model2_history.history['v
al_acc'][-1])
```

```
[Neural Net - Model 1] Loss:  7.79790558079957
[Neural Net - Model 1] Val Loss:  8.034205742033103
[Neural Net - Model 1] Test Loss:  [7.719139232937055, 0.51581027691543
34]
[Neural Net - Model 1] Accuracy:  0.5108695654529828
[Neural Net - Model 1] Val Accuracy:  0.49604743024106085
```

Our initial accuracy isn't great. We achieve an accuracy of 48.9% in the training and 50.4% in the validation, and an accuracy of 48.4% in the test. Let's see if we can improve our network to fit the data better.

In [77]:
```
# create sequential multi-layer perceptron
model3 = Sequential()

# Hidden layers
for i in range(40):
    model3.add(Dense(10, input_dim=input_dim_2,
        activation='relu'))

# output layer
model3.add(Dense(output_dim_2, activation='sigmoid'))


# compile the model
model3.compile(loss='binary_crossentropy', optimizer='adam', metrics=['a
cc'])
model3.summary()
```

```
Layer (type)                 Output Shape              Param #   
=================================================================
dense_50 (Dense)             (None, 10)                150       
_________________________________________________________________
dense_51 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_52 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_53 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_54 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_55 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_56 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_57 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_58 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_59 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_60 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_61 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_62 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_63 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_64 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_65 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_66 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_67 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_68 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_69 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_70 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_71 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_72 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_73 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_74 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_75 (Dense)             (None, 10)                110       
_________________________________________________________________
dense_76 (Dense)             (None, 10)                110       
_________________________________________________________________
```

```
dense_77 (Dense)             (None, 10)                110
_________________________________________________________________
dense_78 (Dense)             (None, 10)                110
_________________________________________________________________
dense_79 (Dense)             (None, 10)                110
_________________________________________________________________
dense_80 (Dense)             (None, 10)                110
_________________________________________________________________
dense_81 (Dense)             (None, 10)                110
_________________________________________________________________
dense_82 (Dense)             (None, 10)                110
_________________________________________________________________
dense_83 (Dense)             (None, 10)                110
_________________________________________________________________
dense_84 (Dense)             (None, 10)                110
_________________________________________________________________
dense_85 (Dense)             (None, 10)                110
_________________________________________________________________
dense_86 (Dense)             (None, 10)                110
_________________________________________________________________
dense_87 (Dense)             (None, 10)                110
_________________________________________________________________
dense_88 (Dense)             (None, 10)                110
_________________________________________________________________
dense_89 (Dense)             (None, 10)                110
_________________________________________________________________
dense_90 (Dense)             (None, 1)                 11
=================================================================
Total params: 4,451
Trainable params: 4,451
Non-trainable params: 0
_________________________________________________________________
```

In [78]:

```
# fit the model
model3_history = model3.fit(
    x_train, y_train,
    epochs=300, validation_split = 0.1, batch_size = 128, verbose=False)
```

In [79]:

```
# model loss
print("[Neural Net - Model 2] Loss: ", model3_history.history['loss'][-1
])
print("[Neural Net - Model 2] Val Loss: ", model3_history.history['val_l
oss'][-1])
print("[Neural Net - Model 2] Test Loss: ", model3.evaluate(x_test, y_te
st, verbose=False))
print("[Neural Net - Model 2] Accuracy: ", model3_history.history['acc']
[-1])
print("[Neural Net - Model 2] Val Accuracy: ", model3_history.history['v
al_acc'][-1])
```

```
[Neural Net - Model 2] Loss:  0.6267417644590524
[Neural Net - Model 2] Val Loss:  0.6291195959220698
[Neural Net - Model 2] Test Loss:  [0.6115785545040026, 0.6432806319398
843]
[Neural Net - Model 2] Accuracy:  0.625857809154077
[Neural Net - Model 2] Val Accuracy:  0.6197530875971288
```

Even after changing hyperparameters, our neural network does not perform very well. Using 40 layers and 300 epochs, the accuracy in the training data is still 62.8% while the accuracy in the test is 65.2%. This is baffling, because we expected our neural network to perform very well. Perhaps this mediocre perforance is due to limitations of our data set (only 14 features and <5000 songs), or of the specific methods we used.

Model Selection

Based upon the presented analysis, we conclude that our boosted decision tree classifier, at a depth of 2 with 751 iterations, is the best model. It achieves the highest accuracy in the test set, of 93.0%.

Moving Forward

We can now try to generate a playlist customized to Grace's taste using our chosen model. We will present the model with a list of songs that both Grace and the model have not seen before. We'll then have the model assess whether these songs should be included in the playlist and then verify that with Grace's opinion.

In [80]:
```
# load in dataset
full_songs_df = pd.read_csv("data/spotify-test.csv")

# drop unnecessary columns
songs_df = full_songs_df.drop(columns=['type', 'id', 'uri', 'track_href'
, 'analysis_url', 'name', 'artist', 'Unnamed: 0'])
```

In [81]:
```
# recreating the best model
best_abc = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2), n_est
imators=800, learning_rate = 0.05)
best_abc.fit(x_train, y_train)
predictions = best_abc.predict(songs_df)
```

In [82]:
```
print("Songs Selected for Grace's Playlist")
for i in range(len(predictions)):
    if predictions[i] == 1:
        print(full_songs_df.loc[i]['name'])
```

```
Songs Selected for Grace's Playlist
Never Seen Anything "Quite Like You" - Acoustic
Crazy World - Live from Dublin
Whatever You Do
Come on Love
1,000 Years
Machine
After Dark
Sudden Love (Acoustic)
Georgia
I Don't Know About You
```

This randomly selected dataset had 26 songs. These songs had never been classified by Grace before and our best model (the boosted decision tree classifier with a depth of 2) was used to predict whether songs would be included in her playlist. We then played all the songs in the dataset to Grace to see whether she would include them in her playlist. The model performed accurately, except for one song which she said she would not have added to her playlist ("I Don't Know About You"). One reason for this mishap could be that our model isn't 100% accurate, so this song could be by chance one of the ones it messes up; 1 missed song out of 26 is reasonable for a model with 93% accuracy. Another reason could be that Grace's actual taste is different from how she made the playlist (perhaps she is in a different emotive or environmental state that temporally affects her preferences, or perhaps her underlying preferences have changed). Despite this error, overall, Grace was pleased that we could use data science to automate her playlist selection procees!