

# Spotify Hit Songs Classification

Nicole Michaud, 8/7/2023

## Business Problem

There could be a number of factors that determine whether a song will be a hit or not. If we know what these factors are, we may be able to better predict how popular a song will be and therefore improve features such as user recommendations.

By looking at Spotify data, can we use certain features to predict whether or not a song will be a hit? What features are most predictive of hit songs?

## Data Understanding

To begin, I import all the necessary tools I will need for this project as well as the dataset I will be working with. I also explore the dataset to get a better understanding of the data it contains.

```
In [1]: 1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 %matplotlib inline
5 import math
6 import seaborn as sns
7 sns.set(font_scale=2)
8 from sklearn.linear_model import LogisticRegression
9 from sklearn.model_selection import train_test_split, cross_val_score,
10 from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
11 from sklearn.metrics import precision_score, accuracy_score, f1_score,
12 from sklearn.tree import DecisionTreeClassifier
13 from sklearn import tree
14 from sklearn.neighbors import KNeighborsClassifier
```

```
In [2]: 1 data = pd.read_csv('data/dataset-of-10s.csv')
        2 data.head()
```

Out[2]:

	track	artist	uri	danceability	energy	key	loudness
0	Wild Things	Alessia Cara	spotify:track:2ZyuwVvV6Z3XJaXIFbspeE	0.741	0.626	1	-4.826
1	Surfboard	Esquivel!	spotify:track:61APOtq25SCMuK0V5w2Kgp	0.447	0.247	5	-14.661
2	Love Someone	Lukas Graham	spotify:track:2JqnpxeIO9dmvjUMCaLCLJ	0.550	0.415	9	-6.557
3	Music To My Ears (feat. Tory Lanez)	Keys N Krates	spotify:track:0cjfLhk8WJ3etPTCseKXtk	0.502	0.648	0	-5.698
4	Juju On That Beat (TZ Anthem)	Zay Hilfigerrr & Zayion McCall	spotify:track:1lItf5ZXJc1by9SbPeljFd	0.807	0.887	1	-3.892

```
In [3]: 1 data.isna().sum()
```

```
Out[3]: track          0
        artist         0
        uri           0
        danceability   0
        energy         0
        key            0
        loudness       0
        mode           0
        speechiness    0
        acousticness   0
        instrumentalness 0
        liveness       0
        valence         0
        tempo          0
        duration_ms    0
        time_signature  0
        chorus_hit      0
        sections        0
        target          0
        dtype: int64
```

In [4]: 1 data.describe()

Out[4]:

	danceability	energy	key	loudness	mode	speechiness	acousticness
<b>count</b>	6398.000000	6398.000000	6398.000000	6398.000000	6398.000000	6398.000000	6398.000000
<b>mean</b>	0.568163	0.667756	5.283526	-7.589796	0.645514	0.098018	0.216928
<b>std</b>	0.191103	0.240721	3.606216	5.234592	0.478395	0.097224	0.296835
<b>min</b>	0.062200	0.000251	0.000000	-46.655000	0.000000	0.022500	0.000000
<b>25%</b>	0.447000	0.533000	2.000000	-8.425000	0.000000	0.038825	0.008533
<b>50%</b>	0.588000	0.712500	5.000000	-6.096500	1.000000	0.057200	0.067050
<b>75%</b>	0.710000	0.857000	8.000000	-4.601250	1.000000	0.112000	0.311000
<b>max</b>	0.981000	0.999000	11.000000	-0.149000	1.000000	0.956000	0.996000

The different features of the data have very different values and will need to be normalized during the data preparation step.

In [5]: 1 data['target'].value\_counts()

Out[5]: 1 3199  
0 3199  
Name: target, dtype: int64

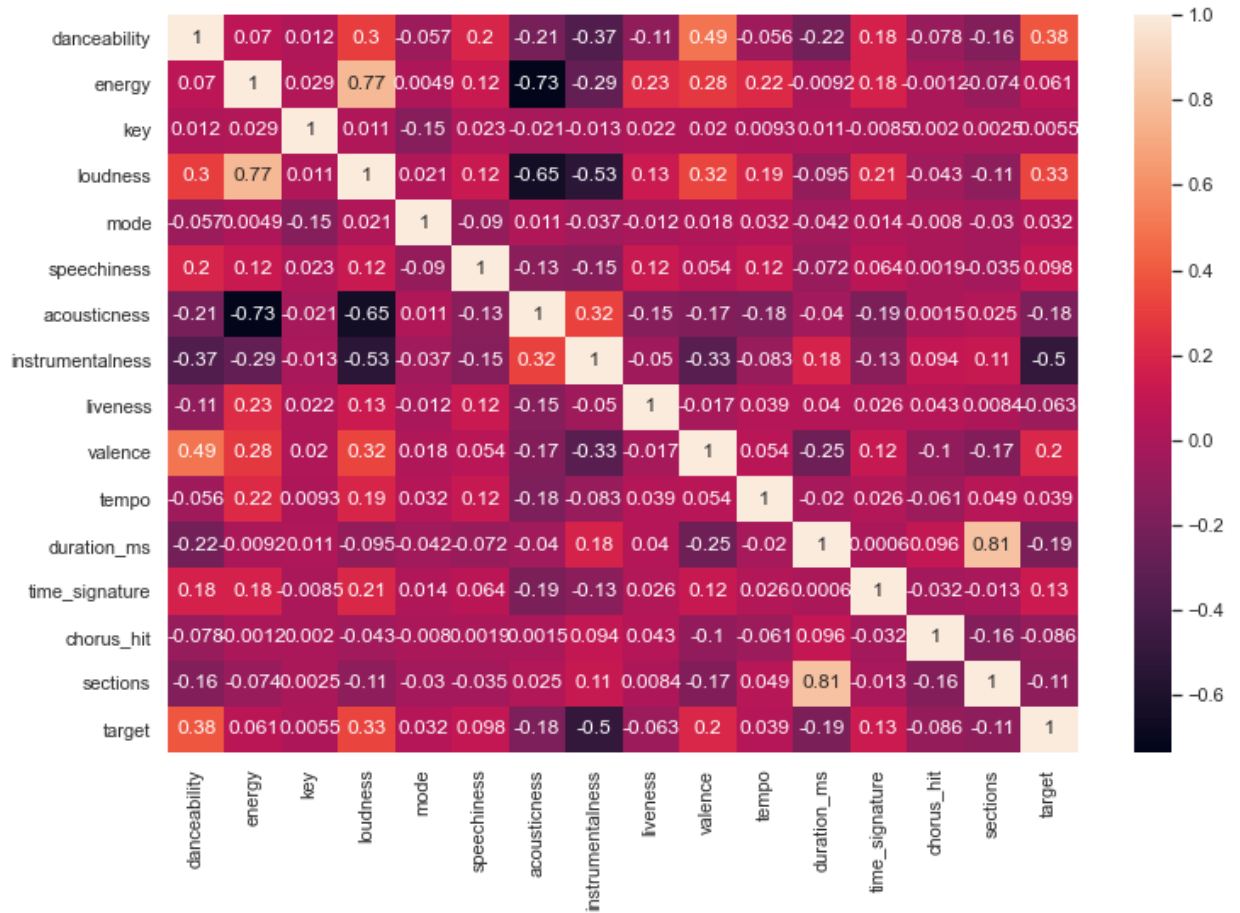
To generally get a better idea of which features of the data might be related to the target, I look at the Pearson correlations.

```
In [6]: 1 # correlation?
        2 data.corr()
```

Out[6]:

	danceability	energy	key	loudness	mode	speechiness	acousticness
<b>danceability</b>	1.000000	0.069645	0.012429	0.300576	-0.057280	0.200090	-0.206865
<b>energy</b>	0.069645	1.000000	0.028703	0.774536	0.004929	0.119194	-0.734853
<b>key</b>	0.012429	0.028703	1.000000	0.010824	-0.146063	0.022796	-0.021271
<b>loudness</b>	0.300576	0.774536	0.010824	1.000000	0.021064	0.122028	-0.648717
<b>mode</b>	-0.057280	0.004929	-0.146063	0.021064	1.000000	-0.090107	0.011424
<b>speechiness</b>	0.200090	0.119194	0.022796	0.122028	-0.090107	1.000000	-0.134226
<b>acousticness</b>	-0.206865	-0.734853	-0.021271	-0.648717	0.011424	-0.134226	1.000000
<b>instrumentalness</b>	-0.371334	-0.288263	-0.013049	-0.533671	-0.037132	-0.148649	0.315563
<b>liveness</b>	-0.107581	0.231393	0.021785	0.126807	-0.011590	0.121075	-0.149926
<b>valence</b>	0.494136	0.281031	0.019572	0.324985	0.018198	0.053836	-0.166253
<b>tempo</b>	-0.056197	0.216886	0.009259	0.194467	0.032180	0.117813	-0.182050
<b>duration_ms</b>	-0.224803	-0.009228	0.011028	-0.094598	-0.042125	-0.071826	-0.039567
<b>time_signature</b>	0.178671	0.175984	-0.008462	0.207436	0.014125	0.063656	-0.191607
<b>chorus_hit</b>	-0.078254	-0.001224	0.001960	-0.042665	-0.007967	0.001857	0.001477
<b>sections</b>	-0.162908	-0.074466	0.002512	-0.111469	-0.030129	-0.035077	0.024583
<b>target</b>	0.384486	0.060701	0.005548	0.327471	0.032021	0.097783	-0.184479

```
In [7]: 1 sns.set(rc={'figure.figsize':(12,8)})
2 sns.heatmap(data.corr(), annot=True);
```



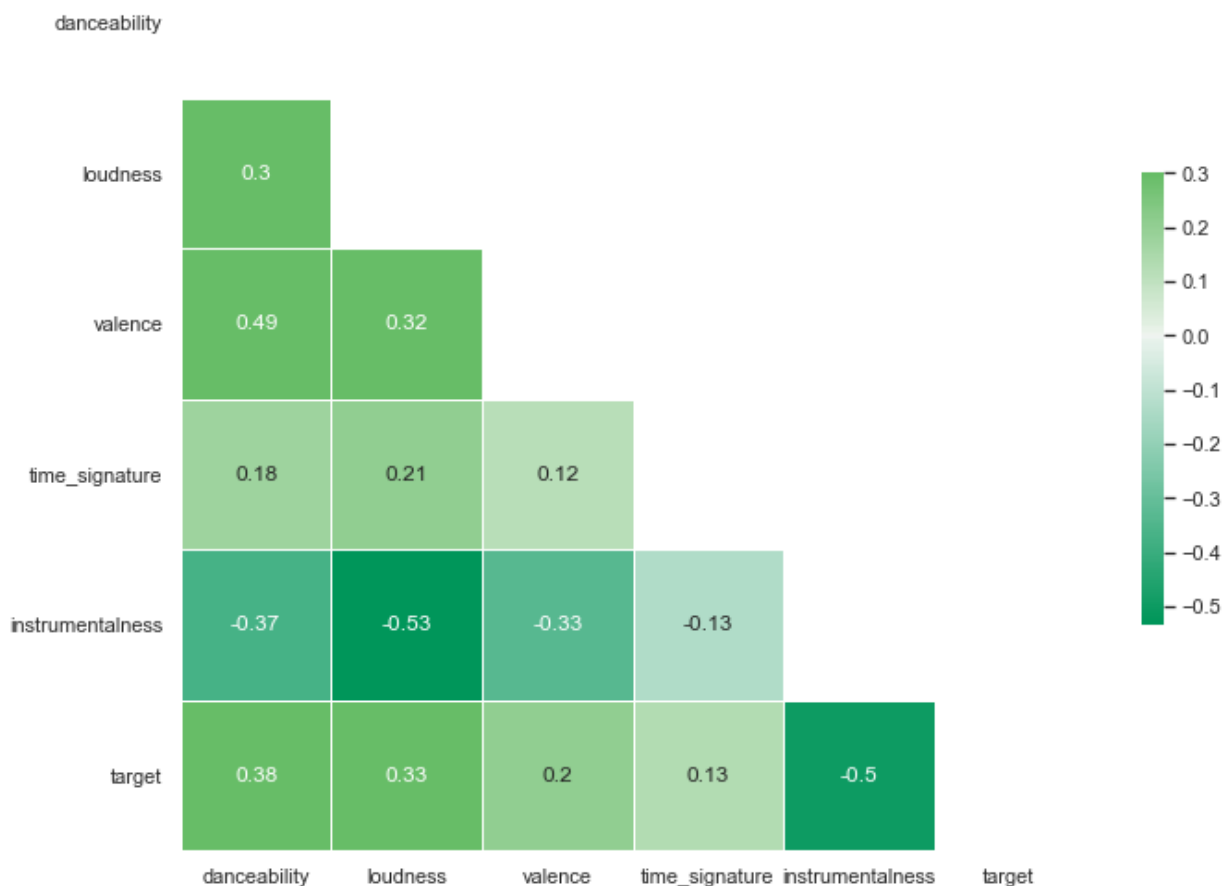
```

In [8]: 1 sns.set(style="white")
2 corr = data[['danceability', 'loudness', 'valence', 'time_signature', '
3
4 mask = np.triu(np.ones_like(corr, dtype=np.bool))
5
6 f, ax = plt.subplots(figsize=(11, 9))
7
8 cmap = sns.diverging_palette(160,120,215, n=3, as_cmap=True)
9
10 sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
11             square=True, linewidths=.5, cbar_kws={"shrink": .5}, annot=
12 plt.show()

```

<ipython-input-8-5b51a6c4189b>:4: DeprecationWarning: `np.bool` is a deprecated alias for the builtin `bool`. To silence this warning, use `bool` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool\_` here.  
 Deprecated in NumPy 1.20; for more details and guidance: <https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations> (<https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations>)

```
mask = np.triu(np.ones_like(corr, dtype=np.bool))
```



Just by looking at this heatmap of the different features' correlations, we can see that the features that are most positively correlated with the target (whether or not the song is a hit) are danceability, loudness, valence, and time signature. Instrumentalness has a strong negative correlation with the

## Data Preparation

To compare later models to, first a baseline model must be created. I am using a single decision tree for the baseline model. In order to do this, I begin by assigning the target variable to y and the other features to X, performing a train-test split to evaluate the model with, and then normalizing the training data.

```
In [9]: 1 X = data.drop(columns=['target', 'uri', 'artist', 'track'], axis=1)
        2 y = data['target']
        3 X.head()
```

```
Out[9]:
```

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness
0	0.741	0.626	1	-4.826	0	0.0886	0.02000	0.000	0.0826
1	0.447	0.247	5	-14.661	0	0.0346	0.87100	0.814	0.0946
2	0.550	0.415	9	-6.557	0	0.0520	0.16100	0.000	0.1080
3	0.502	0.648	0	-5.698	0	0.0527	0.00513	0.000	0.2040
4	0.807	0.887	1	-3.892	1	0.2750	0.00381	0.000	0.3910

```
In [10]: 1 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.2)
```

```
In [11]: 1 #normalize the data
        2 X_train = X_train.apply(lambda x : (x - x.min()) / (x.max() - x.min()),
        3
        4 X_test = X_test.apply(lambda x : (x - x.min()) / (x.max() - x.min()), ax
        5
```

## Modeling

In this section I build and evaluate my baseline decision tree model, then make changes to that model as well as try a more complex model (random forests) to see if it will perform better at correctly classifying the songs as hits or not.

For this business problem, we have a goal of recommending hit songs to Spotify users and therefore it would be better to incorrectly identify a hit song as not a hit, than to incorrectly identify a song as a hit when it isn't one. In other words, a false negative from our model is better than a false positive. We want to only recommend good-quality songs. For these reasons, the metrics used to evaluate the following models include accuracy, precision, and AUC.

```
In [12]: 1 # Building a "regular" tree as a baseline
          2
          3 baseline_tree = DecisionTreeClassifier(criterion='entropy', max_depth=5
          4 baseline_tree.fit(X_train, y_train)
          5
```

Out[12]: DecisionTreeClassifier(criterion='entropy', max\_depth=5)

```
In [13]: 1 y_hat_train = baseline_tree.predict(X_train)
          2 y_hat_test = baseline_tree.predict(X_test)
          3
```

```
In [14]: 1 # Precision score
          2 print("Training Precision for Decision Tree Classifier: {:.4}%".format(
          3 print("Testing Precision for Decision Tree Classifier: {:.4}%".format(p
          4
          5 print("Training Accuracy for Decision Tree Classifier: {:.4}%".format(a
          6 print("Testing Accuracy for Decision Tree Classifier: {:.4}%".format(ac
          7
```

Training Precision for Decision Tree Classifier: 78.93%  
Testing Precision for Decision Tree Classifier: 78.11%  
Training Accuracy for Decision Tree Classifier: 82.76%  
Testing Accuracy for Decision Tree Classifier: 79.81%

```
In [15]: 1 # Check the AUC of predictions
          2 false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,
          3 roc_auc = auc(false_positive_rate, true_positive_rate)
          4 roc_auc
```

Out[15]: 0.7983447170173967

### ***AUC interpretation:***

Because of the fact that for this business problem we would rather have a false negative (fail to identify a song as a hit when it really is one) than a false positive (incorrectly identify a song as a hit), looking at the AUC (Area Under the Curve) can provide us with more information on the rate of false negatives to false positives. We want this value to be as close to 1.0 as possible.

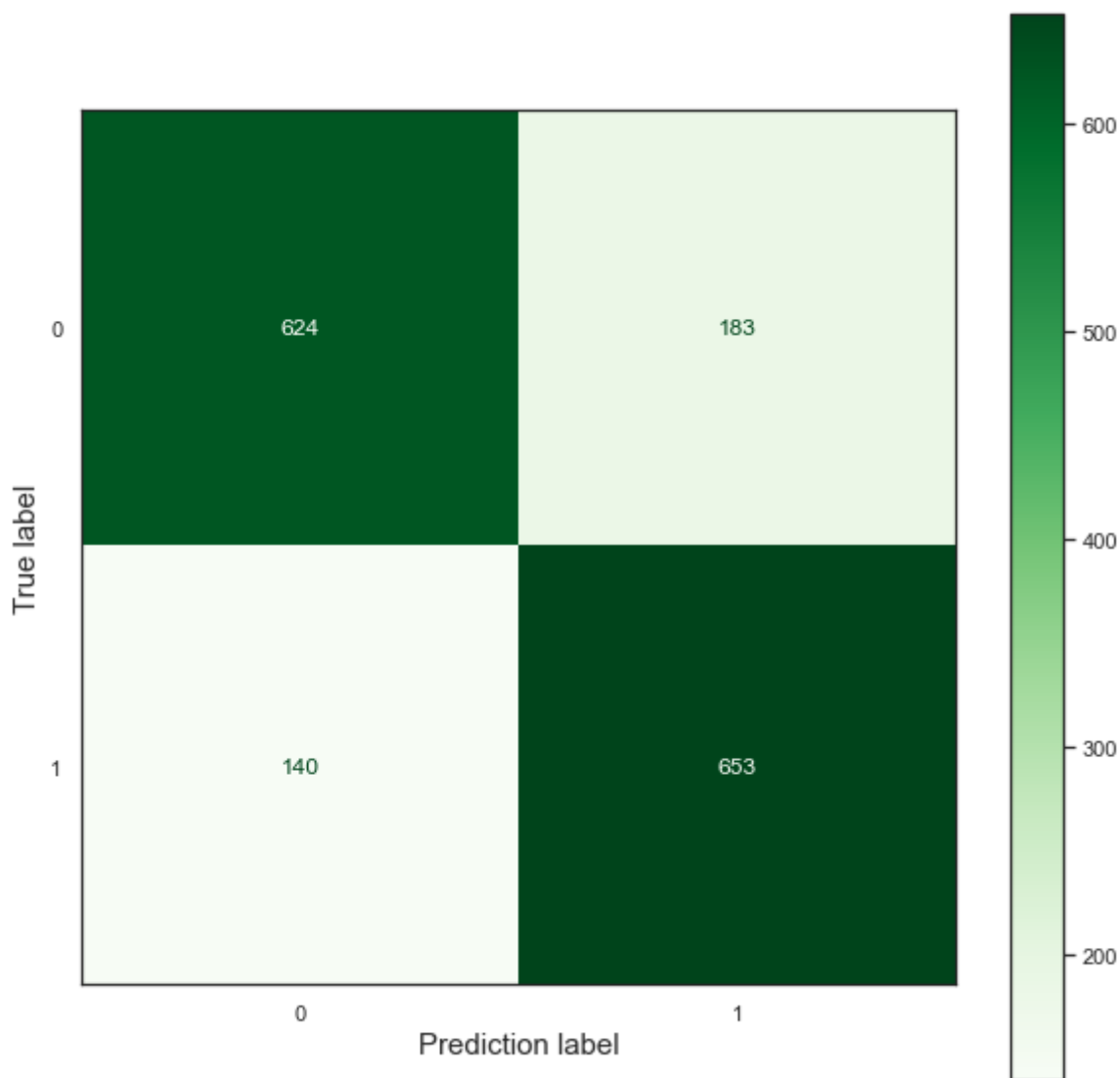
Our baseline model has an AUC value of about 0.80.

```
In [16]: 1 cnf_matrix = confusion_matrix(y_test, y_hat_test)
          2 print('Confusion Matrix:\n', cnf_matrix)
```

Confusion Matrix:  
[[624 183]  
[140 653]]



```
In [17]: 1 #CM Visualization:
2 fig, ax = plt.subplots(figsize=(10,10))
3 cm_1 = ConfusionMatrixDisplay(confusion_matrix = cnf_matrix, display_labels=
4 cm_1.plot(cmap=plt.cm.Greens, ax=ax)
5 plt.xlabel('Prediction label',fontsize=15)
6 plt.ylabel('True label',fontsize=15);
```



### Confusion Matrix interpretation:

This model has 624 true positives, 140 false positives, 653 true negatives, and 183 false negatives. For this data/business problem, it is worse to have a false positive than a false negative and there are less FPs than FNs, which is a good thing.

Next, I examine how important the different features of the data are for this baseline model.

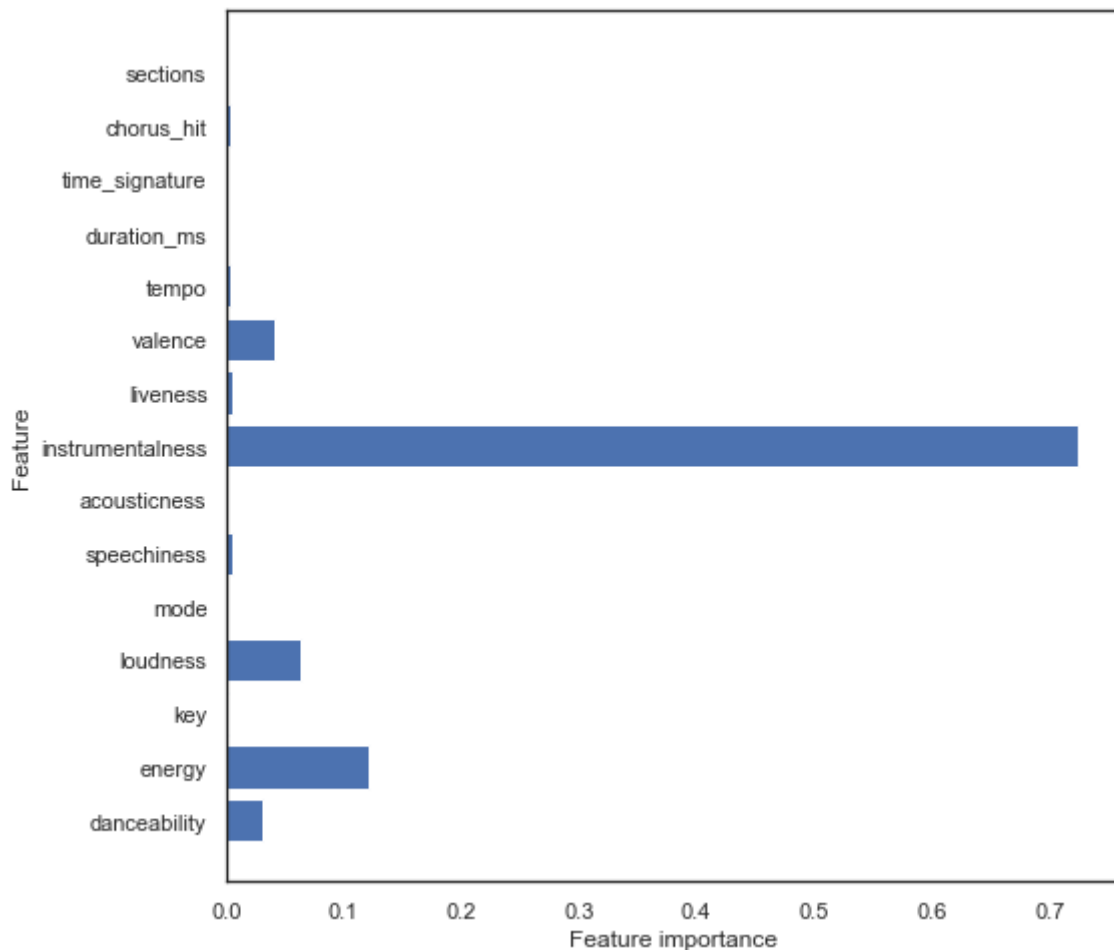
```
In [18]: 1 baseline_tree.feature_importances_
```

```
Out[18]: array([0.03137781, 0.12121346, 0.00222707, 0.06339033, 0.
0.00553143, 0.
, 0.72431934, 0.00508173, 0.04098987,
0.00312003, 0.
, 0.
, 0.00274894, 0.
])
```

```

In [19]: 1 # Plotting feature importances
          2 def plot_feature_importances(model):
          3     n_features = X_train.shape[1]
          4     plt.figure(figsize=(8,8))
          5     plt.barh(range(n_features), model.feature_importances_, align='center')
          6     plt.yticks(np.arange(n_features), X_train.columns.values)
          7     plt.xlabel('Feature importance')
          8     plt.ylabel('Feature')
          9
         10 plot_feature_importances(baseline_tree)
         11

```



The most important features to this baseline model are instrumentalness, energy, loudness, valence, and danceability.

Next, to try and improve this baseline model, I perform a GridSearchCV to search for the optimal combination of parameters that can make our decision tree perform the best and see if it is better than the baseline model's performance.

```
In [20]: 1 dt_param_grid = {
2         "max_depth": [None, 5, 10],
3         "min_samples_leaf": [2, 4, 6, 8],
4         "max_features": [5, 7, 10]
5
6     }
7
```

```
In [21]: 1 # Baseline DT grid search
2 dt_grid_search = GridSearchCV(baseline_tree , dt_param_grid, cv=3, return_train_score=True)
3
4 dt_grid_search.fit(X_train, y_train)
```

```
Out[21]: GridSearchCV(cv=3,
                    estimator=DecisionTreeClassifier(criterion='entropy', max_depth=5),
                    param_grid={'max_depth': [None, 5, 10], 'max_features': [5, 7, 10],
                                'min_samples_leaf': [2, 4, 6, 8]},
                    return_train_score=True)
```

```
In [22]: 1 # Mean training score
2 dt_gs_training_score = np.mean(dt_grid_search.cv_results_["mean_train_score"])
3
4 # Mean test score
5 dt_gs_testing_score = dt_grid_search.score(X_test, y_test)
6
7 print(f"Mean Training Score: {dt_gs_training_score :.2%}")
8 print(f"Mean Test Score: {dt_gs_testing_score :.2%}")
9 print("Best Parameter Combination Found During Grid Search:")
10 dt_grid_search.best_params_
```

Mean Training Score: 87.81%

Mean Test Score: 80.25%

Best Parameter Combination Found During Grid Search:

```
Out[22]: {'max_depth': 5, 'max_features': 10, 'min_samples_leaf': 6}
```

Based on this grid search, the optimal value for minimum leaf sample size is 2, and the optimal maximum tree depth is 5, and the optimal maximum number of features is 7.

```
In [23]: 1 # Train a classifier with optimal values identified above
2 dt = DecisionTreeClassifier(
3         max_depth=5,
4         min_samples_leaf=6,
5         max_features = 7,
6         random_state=42)
7 dt.fit(X_train, y_train)
8 y_pred = dt.predict(X_test)
9 false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,
10 roc_auc = auc(false_positive_rate, true_positive_rate)
11 roc_auc
```

```
Out[23]: 0.7742975634071984
Loading [MathJax]/extensions/Safe.js
```

```
In [24]: 1 y_hat_train2 = dt.predict(X_train)
          2 y_hat_test2 = dt.predict(X_test)
          3
```

```
In [25]: 1 print("Training precision score for decision tree model with tuned hype
          2 print("Testing precision score for decision tree model with tuned hyper
          3
          4 print("Training accuracy score for decision tree model with tuned hyper
          5 print("Testing accuracy score for decision tree model with tuned hyperp
```

Training precision score for decision tree model with tuned hyperparamete  
rs: 78.22%

Testing precision score for decision tree model with tuned hyperparameter  
s: 77.62%

Training accuracy score for decision tree model with tuned hyperparameter  
s: 82.7%

Testing accuracy score for decision tree model with tuned hyperparameter  
s: 77.44%

Changing these parameters did not improve the baseline model.

## Building a random forests model:

```
In [26]: 1 #Fit a random forests model
          2 forest = RandomForestClassifier(max_features='sqrt',
          3                                     max_samples = 0.5,
          4                                     random_state=42)
          5 forest.fit(X_train, y_train)
```

```
Out[26]: RandomForestClassifier(max_features='sqrt', max_samples=0.5, random_state
=42)
```

```
In [27]: 1 y_hat_train3 = forest.predict(X_train)
          2 y_hat_test3 = forest.predict(X_test)
          3
```

```
In [28]: 1 print("Training precision score for random forests model: {:.4}%".forma
          2 print("Testing precision score for random forests model: {:.4}%".format
          3
          4 print("Training accuracy score for random forests model: {:.4}%".format
          5 print("Testing accuracy score for random forests model: {:.4}%".format(
          6
```

Training precision score for random forests model: 95.51%

Testing precision score for random forests model: 85.84%

Training accuracy score for random forests model: 96.71%

Testing accuracy score for random forests model: 76.69%

## Interpretation:

This random forests model had both a higher precision score and a higher accuracy score than our

Now, we create a parameter grid specific to our random forest classifier to try and improve its performance:

```
In [29]: 1
2 rf_param_grid = {
3     "max_depth": [None, 5],
4     "max_features": [5, 7, 10],
5     'min_samples_leaf': [4, 6, 8, 10]
6 }
7
```

```
In [30]: 1 # Now perform the GridSearchCV
2 rf_grid_search = GridSearchCV(forest, rf_param_grid, cv=3)
3 rf_grid_search.fit(X_train, y_train)
4
5 print(f"Testing Accuracy: {rf_grid_search.best_score_ :.2%}")
6 print("")
7 print(f"Optimal Parameters: {rf_grid_search.best_params_}")
```

Testing Accuracy: 84.41%

Optimal Parameters: {'max\_depth': None, 'max\_features': 10, 'min\_samples\_leaf': 6}

```
In [31]: 1 # Train a new random forests model with optimal values identified above
2 rf = RandomForestClassifier(max_depth=None,
3                             max_features = 10,
4                             min_samples_leaf=6,
5                             random_state=42)
6 rf.fit(X_train, y_train)
7 y_train_rf = rf.predict(X_train)
8 y_pred_rf = rf.predict(X_test)
9 false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,
10 y_pred_rf)
11 roc_auc = auc(false_positive_rate, true_positive_rate)
```

Out[31]: 0.8064719017549782

```
In [32]: 1 print("Training precision score for random forests model with tuned hyp
2 print("Testing precision score for random forests model with tuned hype
3
4 print("Training accuracy score for random forests model with tuned hype
5 print("Testing accuracy score for random forests model with tuned hyper
```

Training precision score for random forests model with tuned hyperparamet  
ers: 90.48%

Testing precision score for random forests model with tuned hyperparamete  
rs: 83.52%

Training accuracy score for random forests model with tuned hyperparamete  
rs: 93.29%

Testing accuracy score for random forests model with tuned hyperparameter  
s: 80.69%

Loading [MathJax]/extensions/Safe.js

These metrics are better, however there is still some overfitting. If you change the maximum depth to 5, the AUC score and the precision goes down a little bit, but there is less overfitting.

```
In [35]: 1 rf2 = RandomForestClassifier(max_depth=5,
2                                     max_features = 10,
3                                     min_samples_leaf=6,
4                                     random_state=42)
5 rf2.fit(X_train, y_train)
6 y_train_rf2 = rf2.predict(X_train)
7 y_pred_rf2 = rf2.predict(X_test)
8 false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test,
9 roc_auc = auc(false_positive_rate, true_positive_rate)
10 roc_auc
```

Out[35]: 0.824217791674675

```
In [36]: 1 print("Training precision score for random forests model with tuned hyp
2 print("Testing precision score for random forests model with tuned hype
3
4 print("Training accuracy score for random forests model with tuned hype
5 print("Testing accuracy score for random forests model with tuned hyper
```

Training precision score for random forests model with tuned hyperparamet  
ers: 79.35%

Testing precision score for random forests model with tuned hyperparamete  
rs: 79.0%

Training accuracy score for random forests model with tuned hyperparamete  
rs: 84.45%

Testing accuracy score for random forests model with tuned hyperparameter  
s:82.38%

*This will be our **final model**, with:*

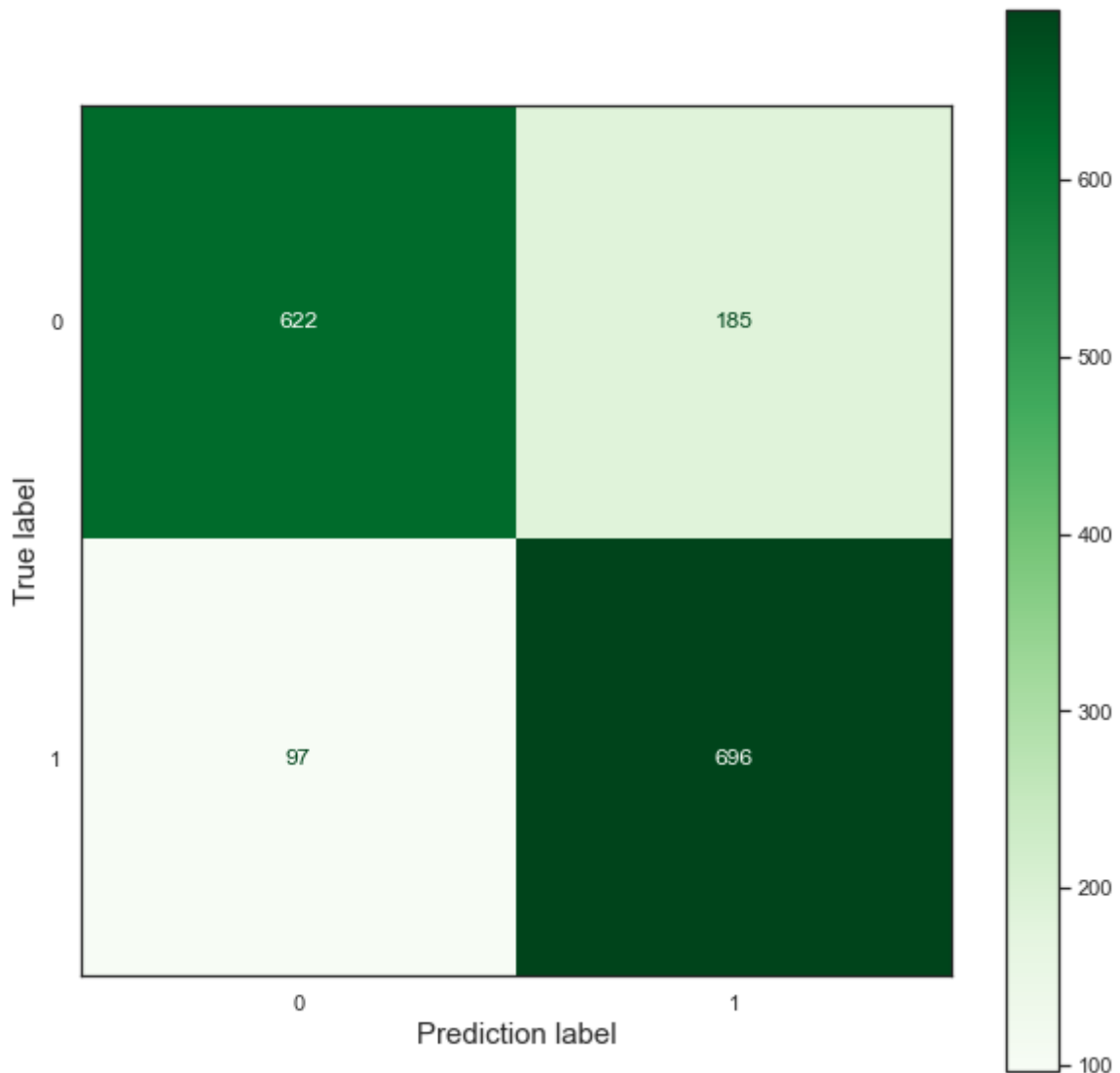
- an AUC score of about 0.82, which is slightly better than our baseline AUC score of about 0.80
- a testing precision score of about 79.0%, which is slightly better than our baseline testing precision of about 78.1%
- a testing accuracy score of about 82.4%, which is slightly better than our baseline accuracy score of 79.8%

```
In [38]: 1 cnf_matrix2 = confusion_matrix(y_test, y_pred_rf2)
2 print('Confusion Matrix:\n', cnf_matrix2)
```

Confusion Matrix:

```
[[622 185]
 [ 97 696]]
```

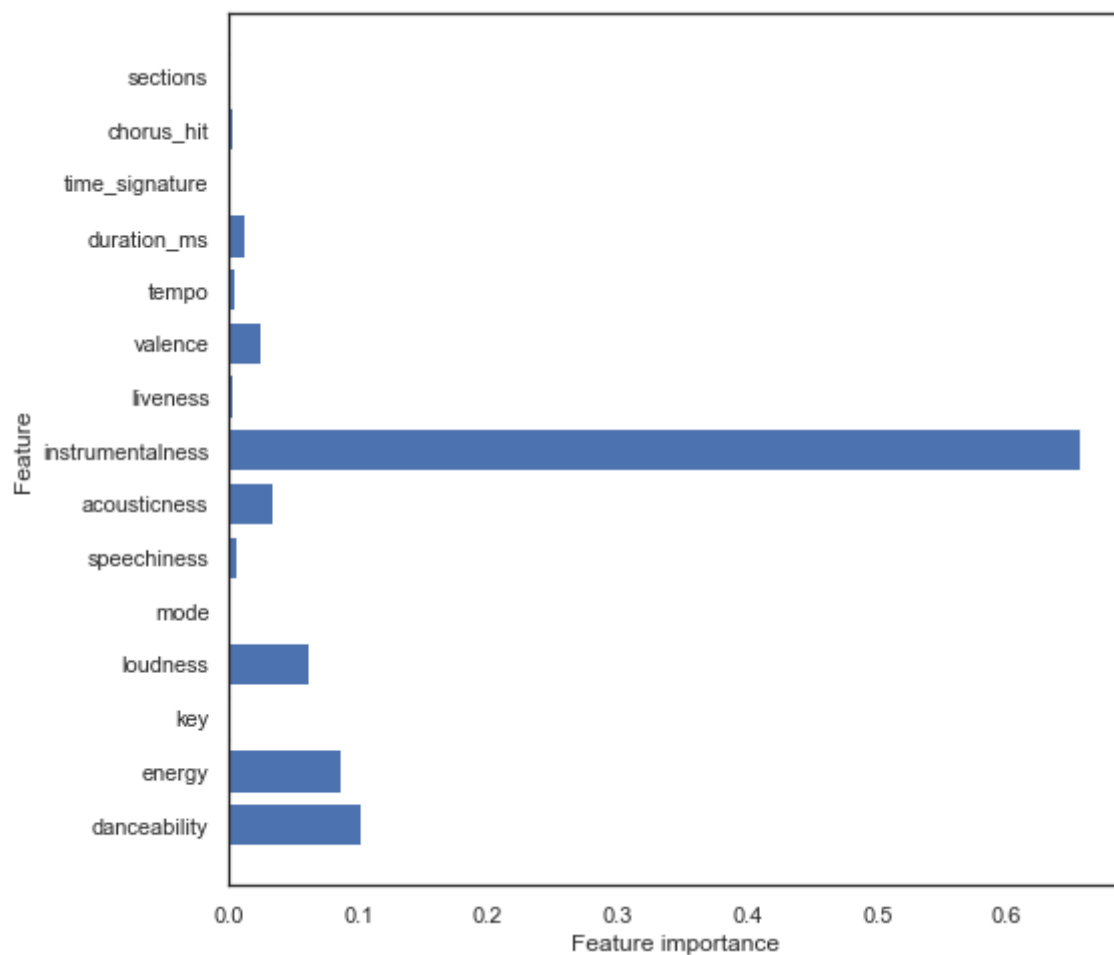
```
In [43]: 1 fig, ax = plt.subplots(figsize=(10,10))
2 cm_2 = ConfusionMatrixDisplay(confusion_matrix = cnf_matrix2, display_1
3 cm_2.plot(cmap=plt.cm.Greens, ax=ax)
4 plt.xlabel('Prediction label',fontsize=15)
5 plt.ylabel('True label',fontsize=15);
6
```



### ***Confusion Matrix interpretation:***

This model has 622 true positives, 97 false positives, 696 true negatives, and 185 false negatives. Once again there are less False Positives than False Negatives, which is what we want. Additionally, there are less of both FPs and FNs than there were in the baseline model.

```
In [40]: 1 # Feature importances of final model:  
2  
3 plot_feature_importances(rf2)
```



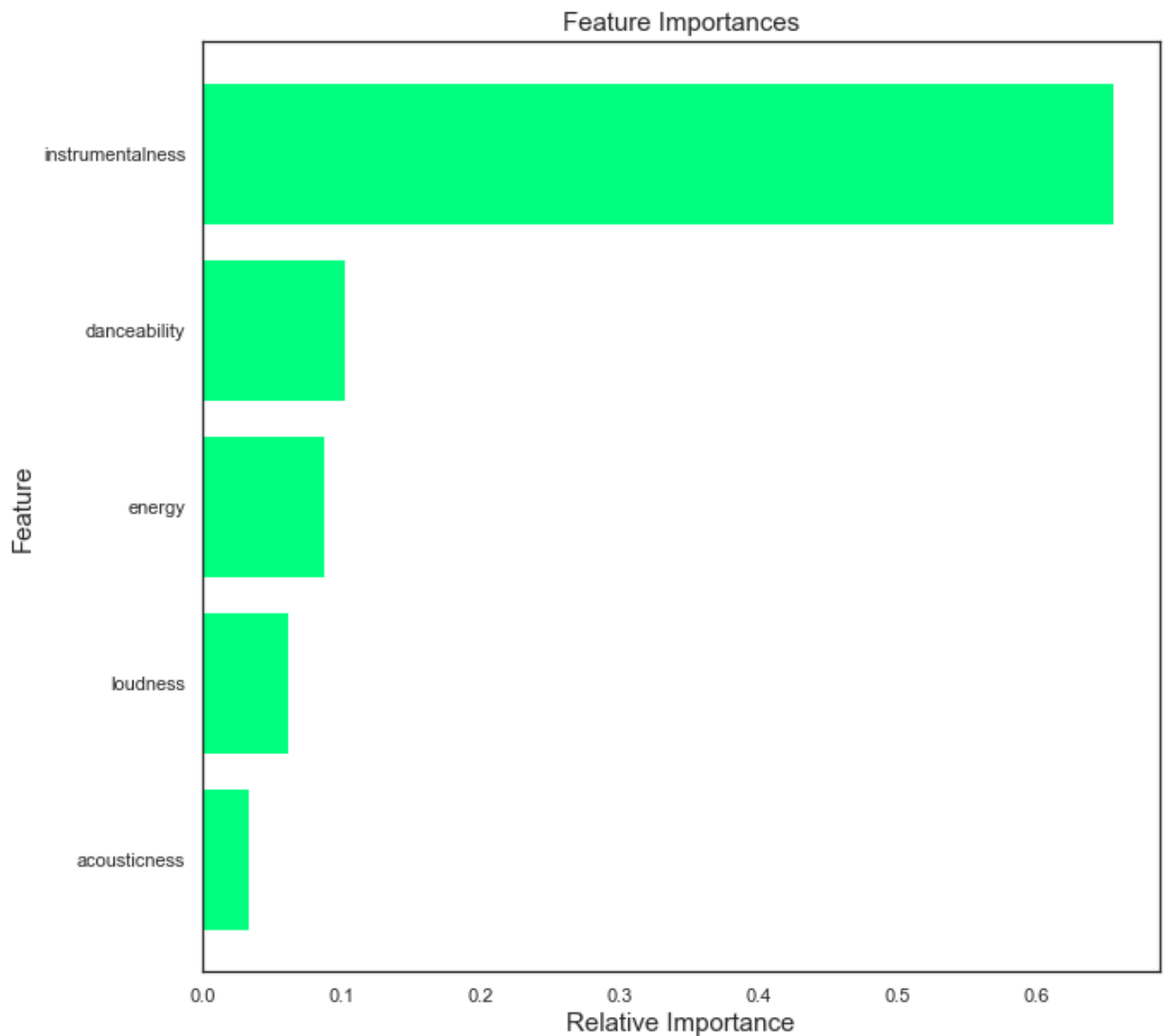
The most important features for this model are instrumentalness, danceability, energy, loudness, and acousticness. These are mostly the same features that were most important to our baseline model, except acousticness is more important here.



```

In [41]: 1 features = X.columns
          2 importances = rf2.feature_importances_
          3 indices = np.argsort(importances)
          4
          5 num_features = 5
          6
          7 plt.figure(figsize=(10,10))
          8 plt.title('Feature Importances', fontsize=15)
          9
         10 plt.barh(range(num_features), importances[indices[-num_features:]], col
         11 plt.yticks(range(num_features), [features[i] for i in indices[-num_feat
         12 plt.xlabel('Relative Importance', fontsize=15)
         13 plt.ylabel('Feature', fontsize=15)
         14 plt.show()

```



## Evaluation

Our final model is about 81.8% accurate and about 77.6% precise in correctly identifying songs as hits or not. The rate of false negatives to false positives (AUC) was about 0.82. This is an improvement to our baseline model which was a single decision tree before hyperparameter tuning

Loading [MathJax] extensions/js/

that was about 80.3% accurate, about 76.8% precise in classifying our target and had an AUC of about 0.81

## Conclusion

### ***Recommendations:***

Hit songs should have the following features:

- Danceability
- Energy
- Loudness
- Acousticness

Songs should not be too high in instrumentalness, because it is highly correlated with a song *not* being a hit.

### ***Next Steps:***

- Compare data from other decades
  - Are there any trends?
- Investigate whether genre is an important feature when determining whether or not a song will be a hit

### ***Context:***

This notebook was created alongside a README file, a miscellaneous notebook, and a slides presentation. All of these can be found on the [GitHub repository](https://github.com/nicolemichaud03/Spotify_hit_songs_classification).  
([https://github.com/nicolemichaud03/Spotify\\_hit\\_songs\\_classification](https://github.com/nicolemichaud03/Spotify_hit_songs_classification)) for this project.