1. Preparing our dataset

These recommendations are so on point! How does this playlist know me so well?

Over the past few years, streaming services with huge catalogs have become the primary means through which most people listen to their favorite music. But at the same time, the sheer amount of music on offer can mean users might be a bit overwhelmed when trying to look for newer music that suits their tastes.

For this reason, streaming services have looked into means of categorizing music to allow for personalized recommendations. One method involves direct analysis of the raw audio information in a given song, scoring the raw data on a variety of metrics. Today, we'll be examining data compiled by a research group known as The Echo Nest. Our goal is to look through this dataset and classify songs as being either 'Hip-Hop' or 'Rock' - all without listening to a single one ourselves. In doing so, we will learn how to clean our data, do some exploratory data visualization, and use feature reduction towards the goal of feeding our data through some simple machine learning algorithms, such as decision trees and logistic regression.

To begin with, let's load the metadata about our tracks alongside the track metrics compiled by The Echo Nest. A song is about more than its title, artist, and number of listens. We have another dataset that has musical features of each track such as `danceability` and `acousticness` on a scale from -1 to 1. These exist in two different files, which are in different formats - CSV and JSON. While CSV is a popular file format for denoting tabular data, JSON is another common file format in which databases often return the results of a given query.

Let's start by creating two pandas `DataFrames` out of these files that we can merge so we have features and labels (often also referred to as X and y) for the classification later on.

In [14]:
```
import pandas as pd

# Read in track metadata with genre labels
tracks = pd.read_csv('datasets/fma-rock-vs-hiphop.csv')

# Read in track metrics with the features
echonest_metrics = pd.read_json('datasets/echonest-metrics.json', precis
e_float=True)

# Merge the relevant columns of tracks and echonest_metrics
echo_tracks = echonest_metrics.merge(tracks[['genre_top', 'track_id']],
on='track_id')
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4802 entries, 0 to 4801
Data columns (total 10 columns):
acousticness        4802 non-null float64
danceability        4802 non-null float64
energy              4802 non-null float64
instrumentalness    4802 non-null float64
liveness            4802 non-null float64
speechiness         4802 non-null float64
tempo               4802 non-null float64
track_id            4802 non-null int64
valence             4802 non-null float64
genre_top           4802 non-null object
dtypes: float64(8), int64(1), object(1)
memory usage: 412.7+ KB
```

In [15]:
```
%%nose

def test_tracks_read():
    try:
        pd.testing.assert_frame_equal(tracks, pd.read_csv('datasets/fma-
rock-vs-hiphop.csv'))
    except AssertionError:
        assert False, "The tracks data frame was not read in correctly."

def test_metrics_read():
    ech_met_test = pd.read_json('datasets/echonest-metrics.json', precis
e_float=True)
    try:
        pd.testing.assert_frame_equal(echonest_metrics, ech_met_test)
    except AssertionError:
        assert False, "The echonest_metrics data frame was not read in c
orrectly."

def test_merged_shape():
    merged_test = echonest_metrics.merge(tracks[['genre_top', 'track_i
d']], on='track_id')
    try:
```

Out[15]: 3/3 tests passed

2. Pairwise relationships between continuous variables

We typically want to avoid using variables that have strong correlations with each other -- hence avoiding feature redundancy -- for a few reasons:

- To keep the model simple and improve interpretability (with many features, we run the
- When our datasets are very large, using fewer features can drastically speed up our computation time.

To get a sense of whether there are any strongly correlated features in our data, we will use built-

In [16]:
```
# Create a correlation matrix
corr_metrics = echonest_metrics.corr()
corr_metrics.style.background_gradient()
```

Out[16]:

	acousticness	danceability	energy	instrumentalness	liveness
acousticness	1	-0.189599	-0.477273	0.110033	0.0413193
danceability	-0.189599	1	0.0453446	-0.118033	-0.143339
energy	-0.477273	0.0453446	1	-0.00241179	0.0457524
instrumentalness	0.110033	-0.118033	-0.00241179	1	-0.0585932
liveness	0.0413193	-0.143339	0.0457524	-0.0585932	1
speechiness	0.0387845	0.171311	-0.00864488	-0.216689	0.0731041
tempo	-0.110701	-0.0943519	0.227324	0.0230032	-0.00756641
track_id	-0.279829	0.102056	0.121991	-0.283206	-0.00405934
valence	-0.0854362	0.428515	0.219384	-0.1452	-0.0178859

In [17]:
```
%%nose

def test_corr_matrix():
    assert all(corr_metrics == echonest_metrics.corr()) and isinstance(c
orr_metrics, pd.core.frame.DataFrame), \
```

Out[17]: 1/1 tests passed

3. Normalizing the feature data

As mentioned earlier, it can be particularly useful to simplify our models and use as few features as necessary to achieve the best result. Since we didn't find any particular strong correlations between our features, we can instead use a common approach to reduce the number of features called **principal component analysis (PCA)**.

It is possible that the variance between genres can be explained by just a few features in the dataset. PCA rotates the data along the axis of highest variance, thus allowing us to determine the relative contribution of each feature of our data towards the variance between classes.

However, since PCA uses the absolute variance of a feature to rotate the data, a feature with a broader range of values will overpower and bias the algorithm relative to the other features. To avoid this, we must first normalize our data. There are a few methods to do this, but a common

In [18]:
```
# Define our features
features = echo_tracks.drop(columns=['genre_top', 'track_id'])

# Define our labels
labels = echo_tracks['genre_top']

# Import the StandardScaler
from sklearn.preprocessing import StandardScaler

# Scale the features and set the values to a new variable
scaler = StandardScaler()
scaled_train_features = scaler.fit_transform(features)
```

In [19]:
```
%%nose

import sys

def test_dropped_columns():
    try:
        pd.testing.assert_frame_equal(features, echo_tracks.drop(columns
=['genre_top', 'track_id']))
    except AssertionError:
        assert False, 'Use the .drop method to remove the genre_top and
 track_id columns.'

def test_labels_df():
    try:
        pd.testing.assert_series_equal(labels, echo_tracks['genre_top'])
    except AssertionError:
        assert False, 'Does your labels DataFrame only contain the genre
_top column?'

def test_standardscaler_import():
    assert 'sklearn.preprocessing' in list(sys.modules.keys()), \
        'The StandardScaler can be imported from sklearn.preprocessing.'

def test_scaled_train_features():
```

Out[19]: 4/4 tests passed

4. Principal Component Analysis on our scaled data

Now that we have preprocessed our data, we are ready to use PCA to determine by how much we can reduce the dimensionality of our data. We can use **scree-plots** and **cumulative explained ratio plots** to find the number of components to use in further analyses.

Scree-plots display the number of components against the variance explained by each component, sorted in descending order of variance. Scree-plots help us get a better sense of which components explain a sufficient amount of variance in our data. When using scree plots, an

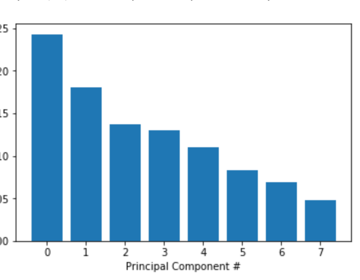
In [20]:
```
# This is just to make plots appear in the notebook
%matplotlib inline

# Import our plotting module, and PCA class
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Get our explained variance ratios from PCA using all features
pca = PCA()
pca.fit(scaled_train_features)
exp_variance = pca.explained_variance_ratio_

# plot the explained variance using a barplot
fig, ax = plt.subplots()
ax.bar(range(pca.n_components_), exp_variance)
ax.set_xlabel('Principal Component #')
```

Out[20]: Text(0.5,0,'Principal Component #')



In [21]:
```
%%nose

import sklearn
import numpy as np
import sys

def test_pca_import():
    assert ('sklearn.decomposition' in list(sys.modules.keys())), \
        'Have you imported the PCA object from sklearn.decomposition?'

def test_pca_obj():
    assert isinstance(pca, sklearn.decomposition.PCA), \
        "Use scikit-learn's PCA() object to create your own PCA object h
ere."

def test_exp_variance():
    rounded_array = np.array([0.24, 0.18, 0.14, 0.13, 0.11, 0.08, 0.07,
 0.05])
    rounder = lambda t: round(t, ndigits = 2)
    vectorized_round = np.vectorize(rounder)
    assert all(vectorized_round(exp_variance) == rounded_array), \
        'Following the PCA fit, the explained variance ratios can be obt
ained via the explained_variance_ratio_ method.'

def test_scree_plot():
    expected_xticks = [float(n) for n in list(range(-1, 9))]
```

Out[21]: 4/4 tests passed

5. Further visualization of PCA

Unfortunately, there does not appear to be a clear elbow in this scree plot, which means it is not straightforward to find the number of intrinsic dimensions using this method.

But all is not lost! Instead, we can also look at the **cumulative explained variance plot** to determine how many features are required to explain, say, about 85% of the variance (cutoffs are somewhat arbitrary here, and usually decided upon by 'rules of thumb'). Once we determine the

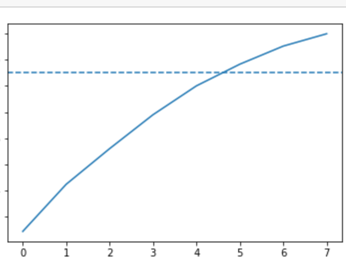
In [22]:
```
import numpy as np

# Calculate the cumulative explained variance
cum_exp_variance = np.cumsum(exp_variance)

# Plot the cumulative explained variance and draw a dashed line at 0.85.
fig, ax = plt.subplots()
ax.plot(cum_exp_variance)
ax.axhline(y=0.85, linestyle='--')

# choose the n_components where about 85% of our variance can be explain
ed
n_components = 6

# Perform PCA with the chosen number of components and project data onto
components
pca = PCA(n_components, random_state=10)
```



In [23]:
```
%%nose

import sys

def test_np_import():
    assert 'numpy' in list(sys.modules.keys()), \
        'Have you imported numpy?'


def test_cumsum():
    cum_exp_variance_correct = np.cumsum(exp_variance)
    assert all(cum_exp_variance == cum_exp_variance_correct), \
    'Use np.cumsum to calculate the cumulative sum of the exp_variance a
rray.'


def test_n_comp():
    assert n_components == 6, \
    ('Check the values in cum_exp_variance if it is difficult '
    'to determine the number of components from the plot.')


def test_trans_pca():
    pca_test = PCA(n_components, random_state=10)
    pca_test.fit(scaled_train_features)
    assert (pca_projection == pca_test.transform(scaled_train_feature
```

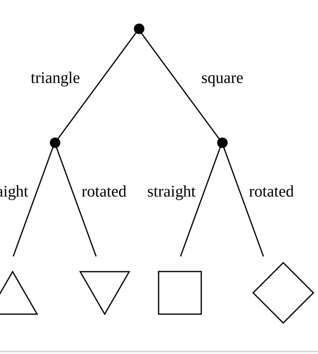
Out[23]: 4/4 tests passed

6. Train a decision tree to classify genre

Now we can use the lower dimensional PCA projection of the data to classify songs into genres. To do that, we first need to split our dataset into 'train' and 'test' subsets, where the 'train' subset will be used to train our model while the 'test' dataset allows for model performance validation.

Here, we will be using a simple algorithm known as a decision tree. Decision trees are rule-based classifiers that take in features and follow a 'tree structure' of binary decisions to ultimately classify a data point into one of two or more categories. In addition to being easy to both use and interpret, decision trees allow us to visualize the 'logic flowchart' that the model generates from the training data.

Here is an example of a decision tree that demonstrates the process by which an input image (in this case, of a shape) might be classified based on the number of sides it has and whether it is rotated.



In [24]:
```
# Import train_test_split function and Decision tree classifier
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

# Split our data
train_features, test_features, train_labels, test_labels = train_test_sp
lit(
    pca_projection, labels, random_state=10)

# Train our decision tree
tree = DecisionTreeClassifier(random_state=10)
tree.fit(train_features, train_labels)

# Predict the labels for the test data
```

In [25]:
```
%%nose

import sys

def test_train_test_split_import():
    assert 'sklearn.model_selection' in list(sys.modules.keys()), \
        'Have you imported train_test_split from sklearn.model_selectio
n?'


def test_decision_tree_import():
    assert 'sklearn.tree' in list(sys.modules.keys()), \
        'Have you imported DecisionTreeClassifier from sklearn.tree?'


def test_train_test_split():
    train_test_res = train_test_split(pca_projection, labels, random_sta
te=10)
    assert (train_features == train_test_res[0]).all(), \
        'Did you correctly call the train_test_split function?'


def test_tree():
    assert tree.get_params() == DecisionTreeClassifier(random_state=10).
get_params(), \
        'Did you create the decision tree correctly?'


def test_tree_fit():
    assert hasattr(tree, 'classes_'), \
        'Did you fit the tree to the training data?'
```

Out[25]: 6/6 tests passed

7. Compare our decision tree to a logistic regression

Although our tree's performance is decent, it's a bad idea to immediately assume that it's therefore the perfect tool for this job -- there's always the possibility of other models that will perform even better! It's always a worthwhile idea to at least test a few other algorithms and find the one that's best for our data.

Sometimes simplest is best, and so we will start by applying **logistic regression**. Logistic regression makes use of what's called the logistic function to calculate the odds that a given data point belongs to a given class. Once we have both models, we can compare them on a few

In [26]:
```
# Import LogisticRegression
from sklearn.linear_model import LogisticRegression

# Train our logisitic regression
logreg = LogisticRegression(random_state=10)
logreg.fit(train_features, train_labels)
pred_labels_logit = logreg.predict(test_features)

# Create the classification report for both models
from sklearn.metrics import classification_report
class_rep_tree = classification_report(test_labels, pred_labels_tree)
class_rep_log = classification_report(test_labels, pred_labels_logit)

print("Decision Tree: \n", class_rep_tree)
print("Logistic Regression: \n", class_rep_log)
```

```
Decision Tree:
              precision    recall  f1-score   support

    Hip-Hop       0.66      0.66      0.66       229
       Rock       0.92      0.92      0.92       972

avg / total       0.87      0.87      0.87      1201

Logistic Regression:
              precision    recall  f1-score   support

    Hip-Hop       0.75      0.57      0.65       229
       Rock       0.90      0.95      0.93       972

avg / total       0.87      0.88      0.87      1201
```

In [27]:
```
%%nose

def test_logreg():
    assert logreg.get_params() == LogisticRegression(random_state=10).ge
t_params(), \
        'The logreg variable should be created using LogisticRegression
().'


def test_logreg_pred():
    assert abs((pred_labels_logit == 'Rock').sum() - 1028) < 10, \
        'The labels should be predicted from the test_features.'


def test_class_rep_tree():
    assert isinstance(class_rep_tree, str), \
        'Did you create the classification report correctly for the deci
sion tree?'
```

Out[27]: 4/4 tests passed

8. Balance our data for greater performance

Both our models do similarly well, boasting an average precision of 87% each. However, looking at our classification report, we can see that rock songs are fairly well classified, but hip-hop songs are disproportionately misclassified as rock songs.

Why might this be the case? Well, just by looking at the number of data points we have for each class, we see that we have far more data points for the rock classification than for hip-hop, potentially skewing our model's ability to distinguish between classes. This also tells us that most of our model's accuracy is driven by its ability to classify just rock songs, which is less than ideal.

To account for this, we can weight the value of a correct classification in each class inversely to the occurrence of data points for each class. Since a correct classification for "Rock" is not more important than a correct classification for "Hip-Hop" (and vice versa), we only need to account for

In [28]:
```
# Subset a balanced proportion of data points
hop_only = echo_tracks.loc[echo_tracks['genre_top'] == 'Hip-Hop']
rock_only = echo_tracks.loc[echo_tracks['genre_top'] == 'Rock']

# subset only the rock songs, and take a sample the same size as there a
re hip-hop songs
rock_only = rock_only.sample(hop_only.shape[0], random_state=10)

# concatenate the dataframes hop_only and rock_only
rock_hop_bal = pd.concat([rock_only, hop_only])

# The features, labels, and pca projection are created for the balanced
 dataframe
features = rock_hop_bal.drop(['genre_top', 'track_id'], axis=1)
labels = rock_hop_bal['genre_top']
pca_projection = pca.fit_transform(scaler.fit_transform(features))

# Redefine the train and test set with the pca_projection from the balan
```

In [29]:
```
%%nose

def test_hop_only():
    try:
        pd.testing.assert_frame_equal(hop_only, echo_tracks.loc[echo_tra
cks['genre_top'] == 'Hip-Hop'])
    except AssertionError:
        assert False, "The hop_only data frame was not assigned correctl
y."


def test_rock_only():
    try:
        pd.testing.assert_frame_equal(
            rock_only, echo_tracks.loc[echo_tracks['genre_top'] == 'Roc
k'].sample(hop_only.shape[0], random_state=10))
    except AssertionError:
        assert False, "The rock_only data frame was not assigned correct
ly."


def test_rock_hop_bal():
    hop_only = echo_tracks.loc[echo_tracks['genre_top'] == 'Hip-Hop']
    rock_only = echo_tracks.loc[echo_tracks['genre_top'] == 'Rock'].samp
le(hop_only.shape[0], random_state=10)
    try:
        pd.testing.assert_frame_equal(
            rock_hop_bal, pd.concat([rock_only, hop_only]))
    except AssertionError:
        assert False, "The rock_hop_bal data frame was not assigned corr
ectly."
```

Out[29]: 4/4 tests passed

9. Does balancing our dataset improve model bias?

We've now balanced our dataset, but in doing so, we've removed a lot of data points that might have been crucial to training our models. Let's test to see if balancing our data improves model bias towards the "Rock" classification while retaining overall classification performance.

Note that we have already reduced the size of our dataset and will go forward without applying any dimensionality reduction. In practice, we would consider dimensionality reduction more

In [30]:
```
# Train our decision tree on the balanced data
tree = DecisionTreeClassifier(random_state=10)
tree.fit(train_features, train_labels)
pred_labels_tree = tree.predict(test_features)

# Train our logistic regression on the balanced data
logreg = LogisticRegression(random_state=10)
logreg.fit(train_features, train_labels)
pred_labels_logit = logreg.predict(test_features)

# compare the models
print("Decision Tree: \n", classification_report(test_labels, pred_label
s_tree))
```

```
Decision Tree:
              precision    recall  f1-score   support

    Hip-Hop       0.77      0.77      0.77       230
       Rock       0.76      0.76      0.76       225

avg / total       0.76      0.76      0.76       455

Logistic Regression:
              precision    recall  f1-score   support

    Hip-Hop       0.82      0.83      0.82       230
       Rock       0.82      0.81      0.82       225

avg / total       0.82      0.82      0.82       455
```

In [31]:
```
%%nose

def test_tree_bal():
    assert (pred_labels_tree == 'Rock').sum() == 226, \
    'The pred_labels_tree variable should contain the predicted labels f
rom the test_features.'


def test_logit_bal():
    assert (pred_labels_logit == 'Rock').sum() == 221, \
```

Out[31]: 2/2 tests passed

10. Using cross-validation to evaluate our models

Success! Balancing our data has removed bias towards the more prevalent class. To get a good sense of how well our models are actually performing, we can apply what's called **cross-validation** (CV). This step allows us to compare models in a more rigorous fashion.

Since the way our data is split into train and test sets can impact model performance, CV attempts to split the data multiple ways and test the model on each of the splits. Although there are many different CV methods, all with their own advantages and disadvantages, we will use what's known as **K-fold** CV here. K-fold first splits the data into K different, equally sized subsets. Then, it

In [32]:
```
from sklearn.model_selection import KFold, cross_val_score

# Set up our K-fold cross-validation
kf = KFold(10)

tree = DecisionTreeClassifier(random_state=10)
logreg = LogisticRegression(random_state=10)

# Train our models using KFold cv
tree_score = cross_val_score(tree, pca_projection, labels, cv=kf)
logit_score = cross_val_score(logreg, pca_projection, labels, cv=kf)

# Print the mean of each array o scores
print("Decision Tree:", np.mean(tree_score), "Logistic Regression:", np.
```

```
Decision Tree: 0.7241758241758242 Logistic Regression: 0.775274725274725
```

In [33]:
```
%%nose

def test_kf():
    assert kf.__repr__() == 'KFold(n_splits=10, random_state=None, shuff
le=False)', \
    'The k-fold cross-validation was not setup correctly.'


def test_tree_score():
    assert np.isclose(round((tree_score.sum() / tree_score.shape[0]),
 4), 0.7242, atol=1e-3), \
    'The tree_score was not calculated correctly.'


def test_log_score():
```

Out[33]: 3/3 tests passed