Classifying Song Popularity on Spotify

By Camelia D. Brumar, Petra Kumi, Philippe Lessard, and Yanniode Peri-Okonny

Problem

The goal of this project was to gain insight into what qualities make songs popular in different parts of the world. More specifically, we aimed to predict the popularity of a song in a certain country based on the features from our dataset. However, due to the significant differences between different popular genres, we believed that predicting the exact position each song would reach on the charts would be too difficult. Instead, we decided to classify the songs into larger categories of popularity in an attempt to increase the accuracy of our predictions. The classes that we chose were: Top 10 (songs ranked 1-10), Top 50 (songs ranked 11-50), Top 100 (songs ranked 51-100), Top 150 (songs ranked 101-150), and Top 200 (songs ranked 151-200).

Data Preparation

In order to do this, we used two different datasets based off of data from Spotify. The first was feature data from Spotify's developer API, which we obtained on Kaggle. Spotify keeps track of many different fields of metadata about every song on the site, such as duration, key, and danceability, the full list of which is listed in Table 1 below. Although the dataset has about 228,000 rows, not much of the data was useable; many songs that fit into different genres were listed multiple times, in rows that were identical, except for the genre.

genre	The genre of the song
artist_name	The name of the artist
track_name	The name of the song
track_id	The Spotify id for the song (can be used to find the song on Spotify)
popularity	The position on the chart
acousticness	Confidence of whether the track is acoustic. Float between 0.0 and 1.0
danceability	Estimate of how suitable the track is for dancing. Float between 0.0 and 1.0
duration_ms	The duration of the track in milliseconds
energy	Measures intensity and activity. Float between 0.0 and 1.0
instrumentalness	Guesses if the track has vocals. Float between 0.0 and 1.0. 1.0 means a track does not contain vocals
key	An estimate of the overall key of the track. Maps pitch classes (C, C#, D, etc) to integers
liveness	Guesses if the track was recorded live. Float between 0.0 and 1.0. 1.0 indicates high probability the track is live
loudness	Average loudness of the track in dB. Float with typical range of -60 to 0
mode	Whether the track is major or minor key. Major = 1, minor = 0
speechiness	Represents how exclusively speech-like the recording is. Float between 0.0 and 1.0. Closer to 1.0 represents more speech-like

tempo	An estimate of the overall tempo of the song in BPM.
time_signature	An estimate of the overall time signature of the track
valence	Represents the positiveness of the track. Float between 0.0 and 1.0. Higher values represent more positive sounding songs

Table 1: Spotify Features Columns

The second dataset we used was Spotify's worldwide daily song rankings, which lists the top 200 most played songs in each of 53 different countries for every day between January 1, 2017 and January 9, 2018. The columns contained in the dataset are listed in Table 2 below. The only columns that we kept from this dataset, however, were the Position column (as the response), the Region column (to indicate which country the song was on the charts in), and the ID column, which was used to merge the datasets. This dataset had over 3.4 million rows.

Position	Position on the charts
Track Name	The name of the song
Artist	Name of the artist
Streams	Number of streams
URL	URL for the song
Date	The date the data is from
Region	The country

Table 2: Spotify Worldwide Song Rankings Columns

In order to make predictions from these two datasets, we had to merge them together and perform feature engineering. This process was done in several steps. First, the irrelevant columns from both datasets were dropped. Then, the track_id column in the features dataset was modified and renamed to match the format of the URL column in the daily rankings dataset. Next, we merged the two datasets into one by doing a left join of the features dataset onto the rankings dataset, using the URL column as a key. For each row in the rankings, if there is a row in the features with a matching URL, the join makes a row in the new dataset containing the combined data from both rows. The next thing we did was remove duplicate rows. This consisted of two steps: first, since some songs were listed multiple times under different genres, we iterated through the rows and flattened these sets of rows into one row with multiple genres. Next, we had to get rid of duplicate songs within the charts. Since the rankings are daily and songs tend to stay popular for relatively long periods of time, most songs were on the charts multiple times in the same or different positions. Since the features of the songs didn't change over time, we decided to use their peak positions (the highest spot they reached on the charts) in order to judge their popularity, and we dropped all rows except the highest for each song in each country. After the merge and removal of duplicate data, we were left with about 12,000 rows of data. There were about 25,000 more songs from the daily rankings that were missing data, and we were able to get most of it by making requests to the Spotify developer API, but we couldn't find genre data anywhere, so we decided to just use the 12,000 data points that we had.

Finally, we performed feature engineering in order to prepare the data for training. First, we one hot encoded genre and region columns. However, the time signature and key required more consideration: after the merge, 5 different values of time signatures appeared: 0/4, 1/4, 2/4, 3/4, and 4/4. For these values, distance has meaning between all but 0/4, which is equally distant from all the others. In order to model this, we plotted the keys around a unit semicircle, with 0/4 at the origin. The keys and modes also required consideration because the same key has a different meaning depending on whether it is major or minor. While there are connections between various major and minor keys, they aren't equivalent, and distance between them is important. Thus, we decided to represent this as two concentric circles. The major key was plotted around a unit circle, and the minor key was plotted around a circle with a radius of 0.5. Finally, we used min-max scaling to scale everything to between 0 and 1 and normalize the data. The final dataset columns are listed in Table 3 below.

Position	The position on the charts. Numerical, and Categorical: 10, 50, 100, 150, or 200
URL	The Spotify URL for the song.
Region	The country the data was recorded in. One Hot Encoded
Acousticness	Confidence of whether the track is acoustic. Float. High value represents high confidence the track is acoustic.
Danceability	Estimate of how suitable the track is for dancing. Float. High value is the most danceable.
Duration_ms	The duration of the track
Energy	Measures intensity and activity. Float. High value is high energy
Instrumentalness	How much vocals the song has. Float. The higher the value, the greater the likelihood the track contains no vocals.

Liveness	Determines if the track was recorded live. Float. High value indicates a strong likelihood that the track is live.
Loudness	Average loudness of the track in decibels (dB). Float
Speechiness	Represents how exclusively speech-like the recording is. Float. High value represents more exclusively speech-like.
Tempo	An estimate of the overall tempo of the song.
Valence	Represents the musical positiveness of the track. Float. Higher values represent more positive sounding songs.
Genre	The genre of the song. One hot encoded
Time_signature	An estimate of the overall time signature of the track. Plotted around a semi-circle, and converted into two columns: time_signature_x, time_signature_y
Key	An estimate of the overall key of the track. Also encompasses the mode of the song. Major key is plotted around the unit circle, and minor key is plotted around a circle with radius of 0.5

Table 3: Final Dataset Columns

Model Selection

Lasso and Ridge Regression

In order to determine which of the predictors were the most impactful and which ones we could discard, one of the methods that was proposed was ridge regression. We also considered using lasso, but lasso turned out to be so strict that none of the predictors were deemed influential enough to be of note. Ridge regression on the other hand did not have this problem. The cross-validation method used to determine alpha was Generalized Cross-Validation, which is described as a form of efficient Leave-One-Out cross-validation. Alpha in this case was 0.035274. After assigning the y-value as the “Position” column of the dataframe and removing the “URL” column, the most influential predictor turned out to be “Region_ee” (Estonia) with a coefficient of -61.53, followed by “Soundtrack” and “Region_sk” (Slovakia). Predictors with coefficients of zero included “Opera”, “A cappella”, and both time signature columns. The calculated error, measured using MSE, was 2903.2298.

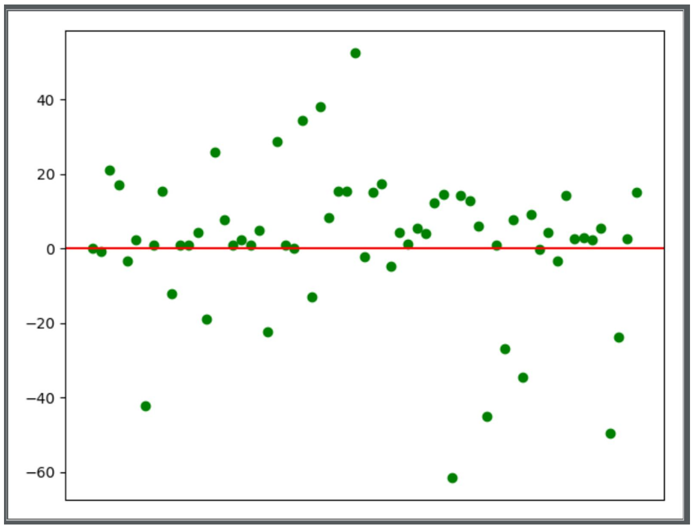

Figure 1: Results of ridge regression on the dataset’s original parameters.

We also ran ridge regression including interaction terms. When we did this under the same parameters as before (aside from alpha now equalling 0.32897) , the most influential parameter was the interaction term “instrumentalness * movie” by quite a bit, as shown in Graph 2. The second most influential parameter was “instrumentalness * Reggaeton”, followed by “instrumentalness *Reggae”. Predictors with coefficients of zero included most of the predictors involving regions interacting with each other, as expected. Other notable predictors higher on the list of most influential terms were “instrumentalness * Region_py” (Paraguay), “instrumentalness * Region_ec” (Ecuador), “instrumentalness* Region_sk” (Slovakia), and “instrumentalness * Region_hk” (Hong Kong) shown in Graph 3. The calculated test MSE was 758,665,632, which is quite high and probably not ideal.

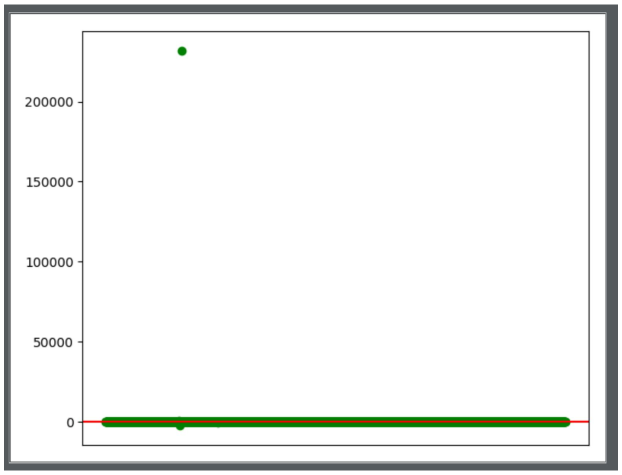

Figure 2: Results of ridge regression on the dataset including interaction terms.

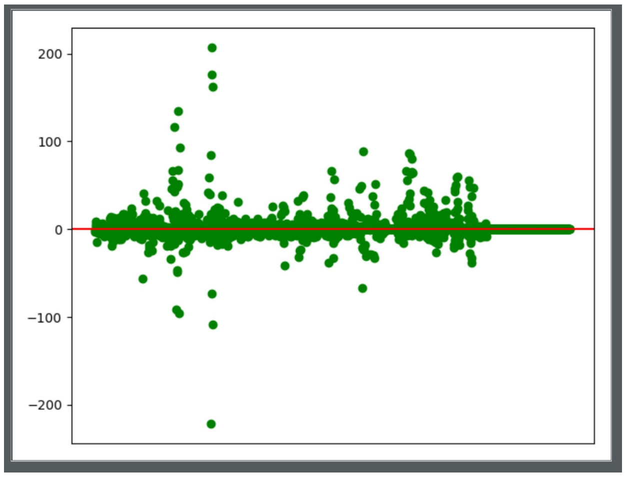

Figure 3: Results of ridge regression on the dataset including all predictors except for the three most influential ones.

K-Nearest Neighbors and PCA

Another method that we tried was k-nearest neighbors. We attempted this method because it did not make many assumptions about the data, and thus was more versatile. To cross-validate this method, we used k-fold cross-validation; the best k with the highest accuracy turned out to be k=10. Also, we noticed that using a bigger k was not positively affecting the accuracy of the method, as seen in the following plot of the 100 attempted k values. We notice that as k increases, the accuracy is decreasing significantly.

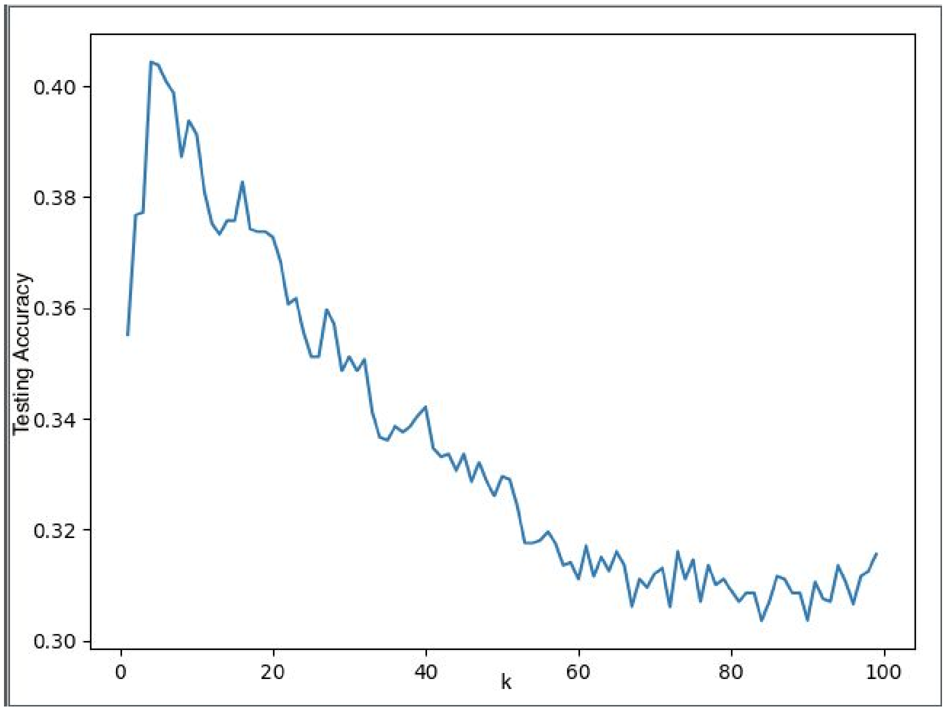

Figure 4: KNN Testing Accuracy over K=1 to K=100

We also attempted to add PCA instead of just taking the given predictors. We chose the number of principal components using k-fold cross validation with 4 folds, and for each number of components we cross-validated the the number of neighbours for KNN. We tested for 1 to 40 principal components, each of which we tested with k values from 1 to 15 in the KNN algorithm. When we did KNN without PCA, we noticed that the best k was 10 with the corresponding accuracy of 0.3852, whereas when performing PCA the best k is 7 with 29 principal components and an accuracy of 0.3859. Using PCA gives us a small improvement in the accuracy, i.e. if we take the difference between the resulting accuracies we only get an improvement of 0.0007.

Random Forest

To implement random forest, we created functions to do the following: iterate through different numbers of trees in the forest, test different numbers of folds for k-fold cross-validation, and test different maximum depths for each tree within the forest. We considered adding 'number of features' as an additional parameter for each tree, but we noticed that the scikit random forest classifier already computes the ideal number of features for us.

First, we cross-validated the optimal number of folds for k-fold cross-validation, using values from k=3 to k=12. For this, we used a smaller number of trees in order to lower the computational intensity. Through multiple rounds of cross-validation - Round 1 using 7-20 trees and Round 2 using 20-30 trees - we learned that in both cases the highest score for the given number of trees occurred for k=6. We also noticed that the more trees there were, the more accurate the model became. This urged us to attempt computations involving larger forests.

After cross-validating the number of folds for k-fold cross-validation, we cross-validated the number of trees that we would need in the forest. When the number of trees in the forest exceeded 28, the computational time became too unreasonable, so we decided to split the dataset into four. The y-value of each data set was whether or not the song was top 10, top 50, top 100, or top 200, respectively. This gave us fewer clusters to deal with, took less time for computations, and higher precision on the testing data.

Now that the function could run much faster thanks to the new datasets, we were able to try a larger number of trees. We initially tried computing accuracy of 100-500 trees with increments of 100, only to see that there was no significant change in accuracy to justify the time results took to ocompute. Hence, we next tried the model with 70-150 trees with 6-fold cross-validation and automatic tree depth. The results of running the function under these parameters showed that the highest accuracy was always at 100 total trees in the forest, so we decided to use that model.

Finally, we decided on the maximum depth of the trees by cross-validating values between 20 and 40. We used these values because we noticed that the built-in scikit function to calculate maximum depth of trees did so by taking the square root of the number of parameters in the dataset. From this, we discovered that a depth of 30 gives us the highest score.

SVM

Since support-vector machines used to be one of the best methods for statistical learning, we decided to try to use it with our dataset. We used 10-fold cross-validation to cross-validate several different parameters from the model. For example, to determine the degree to use for a polynomial kernel, we tested values between 3 and 10 and found that a degree of 3 had the best accuracy. We also needed to determine what kernel to use (Linear, RBF, Polynomial, or Sigmoid), and what to set the C-parameter to. For the C parameter, we tested values of 10^{-5} to 10^{1}, because values above 10^{1} took an unreasonable amount of time for cross-validation. After testing each combination of values with 10-fold cross-validation, we determined that the RBF kernel with $C=10^{1}$ had the highest accuracy. However, we later manually tested larger values of C using the RBF kernel, and found them to be consistently better, with C=10,000 giving the best accuracy while still taking a reasonable amount of time (given that it was not running in a loop). The RBF kernel with C=10,000 had about 54.2% testing accuracy. More detailed testing results are shown in Figure 5 below.

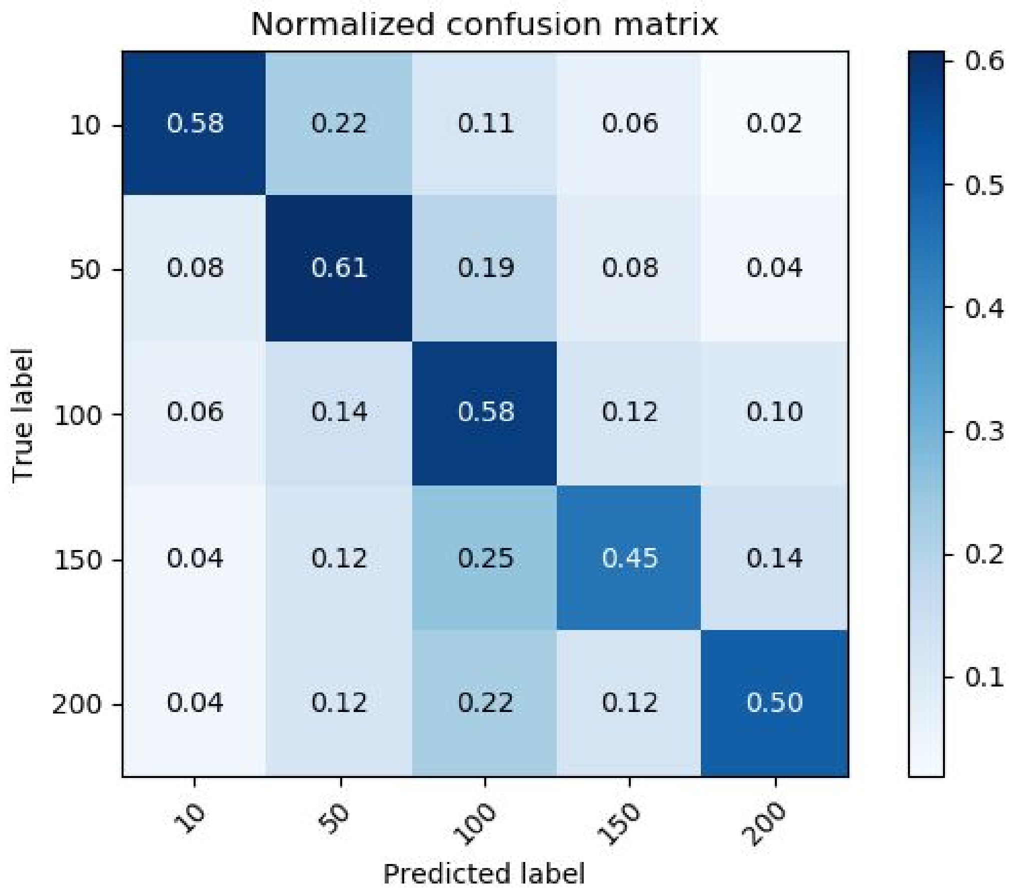

Figure 5: Normalized Confusion Matrix for Multiclass SVM Model

We tried two different strategies for using SVM. One was using a multi-class SVM algorithm, and the other was a layered set of one-vs-all binary classifiers. We separately trained and cross-validated 4 different SVM models: one for Top 10 vs Rest, one for Top 50 vs Rest, one for Top 100 vs Rest, and one for Top 150 vs Rest. Then, we took our x values and ran them through the Top 10 model. Everything that was classified as rest, was then run through the Top 50 model, and so on. This continued until we reached the Top 150 model, which classified all the remaining points. These predictions were then combined to create a full multi-class prediction for the testing set. However, when training these models the sample sizes of the different classes were usually different, which led to bias in the training. In order to mitigate this, we up-sampled the smaller class to the size of the larger class by bootstrapping. This allowed the class samples to be balanced, while not losing data by down-sampling. After testing, however, the single multi-class SVM model consistently had higher testing accuracy, so we chose that as our final SVM model.

Summary of the Final Approach

After cross validating to find the optimal number of hyperparameters for the models described above, we tested their accuracy on sets of testing data with the goal of finding the most successful model. We noticed that a single multi-class random forest with 100 trees, 6 folds, and 30 as our maximum depth for each tree gave us the best accuracy at 0.7219438877755511, followed by a single multi-class SVM, and KNN with 10 neighbors and 29 PCA components. We decided to use the model with the highest testing accuracy, random forest, as our final one, taking into account the fact that unlike the other models, random forests do not overfit. We saved this model and proceeded to test it against the data we had put in the safe.

Results

We ran our predictive model against the safe data and received unexpected results. Based on our success with testing on bootstrapped data, we were expecting around 70% accuracy. Instead, the model only predicted with 37.4% accuracy. The confusion matrix below shows our model's accuracy on each class, starting with 11% True Positives for predicted top ten songs, 53% True Positives on top 100 data, and continuing further with 31% of the data points rightfully classified as top 200.

Ideally, one would want to see high values on the diagonal of a confusion matrix, and really small or zero values on the rest of the matrix. In this case, we receive highest prediction accuracy for top 100, and lowest for top 200. Our model does well in predicting the general trends of a song, confined to whether it might hit the top 100 charts regardless of whether it is placed first or 99th, or to whether it will land on the bottom 100 of the charts, again disregarding more precise classification. However, looking at the data trends, it seems like the model could have been trained to more accurately categorize top 10 songs in their rightful category instead of top 50 or top 100. The same goes for top 200 data.

Looking back at the possible reasons behind our surprisingly low model accuracy, we suspect to have accidentally data snooped. We were always attentive to ensure that we were always testing on some subset of our data, be it through bootstrapping or k-fold cross-validation. However, this was collectively our first attempt at data analytics with Python, so it is highly possible that we might have overlooked or misunderstood how some functions are used. Another explanation for these low accuracy rates could be the fact that we were only looking at average accuracy when deciding on our final model, without taking into account the amount of true and false positives and negatives. Seeing that we trained a model to fit all the clusters of classification, some of which had more data points than others, we might have also picked a more biased model that is much more accurate on true negatives than it is on true positives.

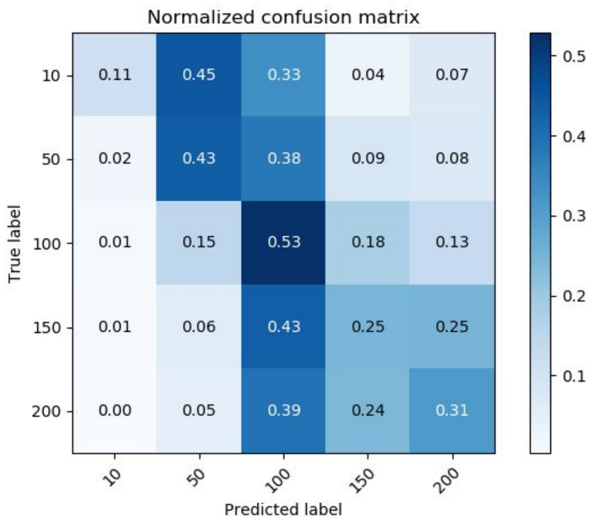

Figure 6: Normalized Confusion Matrix for Final Random Forest Model

Conclusions

In conclusion, we tried to fit 6 different types of models to our data: Lasso, ridge regression, KNN with PCA, KNN without PCA, Random Forest, and SVM. This was done with the purpose of comparing our models to find the one with the highest accuracy. Before proceeding any further, we merged Spotify's Developer API with Spotify's worldwide daily song rankings to obtain a dataset of both rankings of a song in the charts and its features (genre, tempo, mode, liveliness, danceability etc.). We used the URL column as the key to join these datasets.

Next, we removed entries with duplicate or missing data, followed by doing feature engineering to make the dataset useable for our purposes: we one-hot encoded the genre and column regions, represented tempo using the unit circle with 0/4 at the origin (representing songs that change tempo), and represented major and minor keys using two concentric circles with an initial radius of 0.5. Finally, we used min-max scaling to normalize our data, and cross validated the radius of the concentric circle representing minor keys. We then proceeded to store some of the data in "the safe" to only use for testing of the final model. When all this was completed, we could move on to evaluating the accuracy of the selected models. We used k-fold cross-validation and bootstrapping to create testing and training datasets, then tried fine-tuning our models by cross validating on different hyperparameters. The most accurate model was the random forest with 100 trees, of 30 maximum features each, and 6 folds for k-fold cross-validation. After training our final model, we saved it and ran it on our safe data, for which we got a much lower accuracy than expected. Given than overfitting is not an issue in Random Forests, we suspect to have accidentally data snooped or not have examined model accuracy scores in more depth, leading to an overly biased model.

Throughout the process, the most time-consuming task was by far cross-validating our models. Before splitting datasets so each of them only contained two categories , top-x or not top-x, running cross-validation functions on large numbers was too computationally expensive, forcing us to resort to smaller hyperparameters. However, the process became easier and faster once we got used to it and finally generated all the necessary datasets. Out of all the models, the one which took most time and effort was random forest, as it had many modifiable parameters that required multiple rounds of cross-validation.