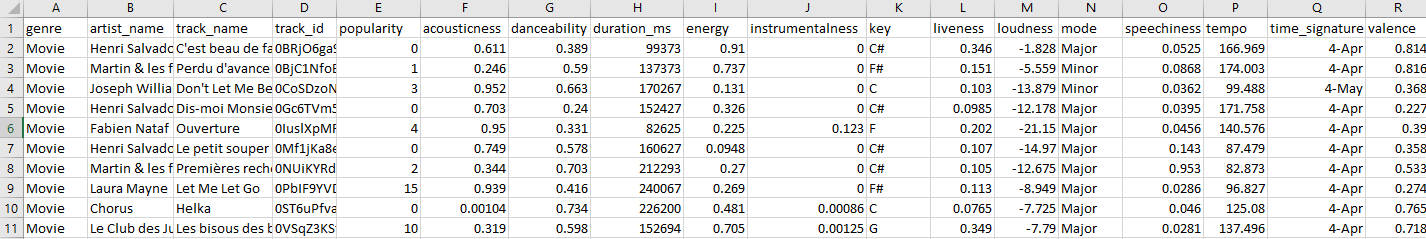
**Spotify Popularity**

For this project, we are using a dataset from Kaggle named **SpotifyFeatures.csv**. Our objective is to predict the popularity of a song given the following features:



**Repo Breakdown**

This section explains the use of some files in the repo that were needed to do this project.

**SpotifyFeatures.csv:** The file straight out of Kaggle that has all features and label column.

**transformation.py:** This is the file that did all the data transformation mentioned here as well as saving the final results to a separate csv file.

**transformed\_data\_one\_hot\_genre-key.csv:** The final dataset to use for training after all transformation steps have been applied.

**trainer.py:** The python script that does the *actual* training and evaluating of the models. In order to use, simply enter the following command: `python3 trainer.py`

**Data Transformation**

In its current state, the data is not usable. Some transformations must be made. First of all, there are some columns in this dataset that serves no purpose.

**Dropped Columns:** Artist\_name, track\_name, track\_id.

These columns are dropped because they are mostly unique. No pattern can be learned from columns that do not have repeating values, or are not ordinal in any way.

This leaves us with the following feature columns: genre, acousticness, danceability, duration\_ms, energy, instrumentalness, key, liveness, loudness, mdoe, speechiness, tempo, time\_signature, and valence. Additionally, one of the columns is actually the target variable, or **label:** popularity.

Some of these columns are still not in usable conditions, so some additional transformations must be made.

**Standardization:** duration\_ms, loudness, and tempo.

These three columns have numeric values in them that vary by a lot. In order to be more usable by our algorithm, we must *standardize* these values. This means, we make the mean value 0, and replace every entry in the column for it’s Z-score on a standard scale.

**Binarization:** Mode

This column only has two possible values, Major or Minor. Because of this, we just need to replace Minor for 0s, and Major for 1s. This makes this column usable.

**Ordinal Transformation:** time\_signature.

Time signature has 5 possible values: 0/4, 1/4, 3/4, 4/4, and 5/4. Because time signature represents how many measures in a bar, there is some information learned from retaining some of the hierarchy of the speed of this feature. This is why we replace each value for its decimal counterpart: 0, .25, .75, 1, and 1.25 respectively.

**One-hot Encoding:** genre and key.

Finally, we arrive at the columns that are *nominal* or also known as *categorical.* Because our algorithm requires all data to be numerical, some transformation has to be made on all strings—such as these two columns. This means, we cannot simply replace these columns with numbers carelessly. Simply replacing these values by arbitrary numbers may cause the algorithm to fixate on the unintentional ordering of the value (for example, if **Movie** genre is given a 1, and **Rap** category is given a 2, the algorithm may learn to believe that **Rap** is twice of a **Movie,** when in reality, this association does not make sense). There must be a more specialized approach. One-hot encoding means we create a brand-new column for each unique entry in the nominal column in question. In this case, for genre we end up with 29 features instead of 1, and for key we end up with 12. And now all there is left to do is to simply place a 1 on the column created with the original entry’s name on it, and zero out the rest. For instance, if **A#** used to be the entry in the key column, instead, there will be a 1 under the **A#** *column,* and a 0 on the other 11. This allows the algorithm to learn certain patterns associated with this key.

Once all these transformations have been made, we have saved the resulting data to a file named **transformed\_data\_one\_hot\_genre-key.csv** to save us time from having to recompute this multiple times. Now that our data is completely transformed, we may now begin to experiment with certain algorithms.

**Selecting a Model**

There are two different machine learning approaches we can take to calculate the popularity of a song in this case: **classification** or **regression.** We have tried each of these two with their own set of parameters to yield different results as described in the following sections. For all of our models, we chose to use **Random Forests** as our learner because its ensemble approach means overfitting will not be an issue.

**Model 1: Binomial Classification**

The first approach taken was binomial classification. This means, we must separate our labels into two categories: popular, and not popular. This means that we needed a *threshold* parameter to determine after what popularity a song is deemed popular vs not popular. We tried many thresholds, but chose to focus on 2 of them.

**Popularity > 44:** Our first threshold is 44. This means that any song with a popularity greater than 44 is considered popular. The reason this threshold is meaningful is because this divides our entire dataset into almost equal 0s and 1s labels, thus being the most balanced.

**Popularity > 70:** The second threshold we focused on was 70. This seemed like a sensible point at which a song may be considered popular. This, however, meant that our data split would end up really imbalanced. To be more exact, 96.2% of our labeled data are now 0’s, and the remaining 3.8% are 1’s. Because of this imbalance, we decided to include F1 calculations on our scoring to focus on the *true positive* prediction power of our model. This ended up giving terrible performance.

**Model 2: Multinomial Classification**

The second approach we took was to change the amount of labels each song could fall into. Instead of popular vs not popular, this approach is more like not popular, not very popular, somewhat popular, popular, very popular, etc. We tried 3, 5, and 10 categories. As one might expect, the more categories we split our songs into, the less successful the algorithm became.

**Model 3: Regression**

Finally, we decided to attempt regression. This means that the label, rather than discrete, is now a continuous value. Our algorithm is basically attempting to determine exactly how popular a song is going to be given the same set of features.

**Results**

The following results were all computing using **sklearn.metrics.** These are the results after all the data had been transformed as described above, and using **max\_depth=100** and **n\_estimators=200** as the models’ hyperparameters.

|  |  |  |
| --- | --- | --- |
| MODEL | CLASSIFIER SCORE | F1 SCORE |
| Binomial ( > 44) | 87.07% | 86.44% |
| biNOMIAL ( > 70) | 96.24% | 2.78% |
| MULTINOMIAL (3 CLASSES) | 79.77% | N/A |
| MULTINOMIAL (5 CLASSES) | 66.74% | N/A |
| MULTINOMIAL (10 CLASSES) | 45.77% | N/A |
| REGRESSION | 74.18% | N/A |

**Note:** Even though the second model has a really high classifier score, it has an abysmal F1 score. This is likely due to the imbalance splitting at 70 popularity causes, which means that the model learns to predict 0 too often, which means high accuracy, but terrible precision/recall.