# Udacity Machine Learning Engineer Nanodegree

# Capstone Project Report

**Mark Bannister**
*January 2018*

## I. Definition

### Introduction

The advent of the Internet has fundamentally changed the way we consume music [1]. Never before has it been so easy to access such immense libraries of music, or to obtain recommendations on what we should be listening to next. These recommendations have evolved significantly, from reviews of major releases, to niche interest music blogs, to (increasingly) personalised recommendations based on our listening habits [2].

The downside to all this information is that it can be increasingly difficult to find the signal within the noise. With only a finite amount of listening time at our disposal, how can we best expose ourselves to new music that we are more likely to enjoy? Put more simply, could we predict the probability of enjoying any given song based on our listening habits, such that we could prioritise our new listening accordingly?

### Problem Statement

As a passionate music fan, I love listening to new music and frequently take advantage of services like Spotify's Discover Weekly. However, some weeks I am unable to listen to all (or any) of my recommendations, and would like to develop a means to assess which of the recommended tracks I am most likely to enjoy. This way I could either listen to only those tracks that I had time for, or save the most likely tracks to be listened to at a later date.

This could be achieved by analysing my music listening history to determine whether there were any particular musical properties shared by tracks that I listen to more (or less) – a proxy for enjoyment. A general model could therefore be developed that, when presented with the musical properties of a new track, was able to predict whether I was likely to enjoy the track or not.

### Previous Work

Machine learning in music is an area of interest for many researchers and companies. Much work has been done on predicting musical genre from audio files, including by Bergstra et al (2006), who used the AdaBoost algorithm to predict one of 10 genres with 83% accuracy [3]; by Indraprastham et al (2017), who achieved 65-83% accuracy using Support Vector Machines [4]; and by Despois (2016), who used convolutional neural networks to predict one of six genres, with 90% accuracy [5].

All of these approaches used some type of spectral analysis to generate predictive features from audio waves. Dieleman (2014) used similar techniques to develop a content-based music recommendation system while interning at Spotify [6]. This approach used convolution neural networks to identify similar songs given an input track, and was proposed as a means of suggesting other tracks that a user might enjoy.

At least two researchers have explored whether it is possible to predict the likely enjoyment of any given song using higher level musical features, such as those made available by Spotify [7]. Leclerq (2016) and Hipolito (2017) both trained classification models on manually-labelled lists of 'liked' and 'disliked' tracks, using predictive features obtained from Spotify [8], [9]. Hipolito reported a test accuracy of 85.16%, precision of 86.06% and recall of 86.06%. Leclerq did not report performance metrics.

I will take a similar approach in this investigation, albeit using a larger dataset, and with track enjoyment (i.e. 'like' / 'dislike') inferred from total play count, rather than manual labelling.

**Datasets and Inputs**

The dataset being considered for this project consists of over 11 years of my personal music listening history, as recorded by Last.fm (an internet radio service). The dataset consists of each unique track I have listened to since July 2006, together with listening data from Last.fm (date first played, date last played and total number of plays), and musical analysis data from Spotify.

Using a simple web app [10], I downloaded my entire Last.fm play history to a csv file and used Pandas to wrangle the data into a usable format. I then used the Spotipy Python library to obtain Spotify's musical analysis of each track [11]. This analysis includes traditional musical features like tempo, loudness, key and tonality, as well as idiosyncratic features like 'energy', 'speechiness' and 'valence', as developed by Spotify and The Echo Nest (a Spotify subsidiary) [7].

In total, the dataset consists of 10,832 tracks and 15 potential predictor variables.

It should be noted that, while this is the most detailed and relevant dataset available for the project, it is subject to inherent bias. It does not include songs that I chose not to listen to in the first place (i.e. selection bias), possibly following my exposure to them in other (non-tracked) contexts, e.g. in public places or on the radio.

It may also be the case that my tastes will have changed to such a degree over time that songs I enjoyed in 2008 look nothing like songs I enjoyed in 2016, potentially introducing a high degree of noise into the data.

Finally, many of the songs in this dataset will have been recommended to me by Spotify based on my listening history, which may involve deeper musical analysis than can be achieved using this project dataset [6]. This will probably introduce a further source of selection bias.

Taken together, these factors mean that this dataset should really be considered a collection of the songs I either already enjoy, or thought myself likely to enjoy (either by my own reckoning or via Spotify's recommendation system). Therefore, rather than separating 'liked' from 'disliked' tracks, it would perhaps be more accurate to suggest that I am seeking to separate 'most-liked' from 'less-liked' tracks.

**Solution Statement**

I will use machine learning techniques to build a music classification model that separates tracks into probable 'likes' and 'dislikes' based on a musical analysis of my listening history.

A ‘liked’ track will be defined as one that I have played three or more times – a slightly crude metric, but a reasonable proxy for enjoyment given I will have specifically sought to play the track at least two additional times following my first listen – something I am not likely to do by accident. Correspondingly, tracks that have been played two times or less will be labelled as ‘disliked’.

The model will be trained to use variables like tempo, key and loudness to predict the label of a song.

If the model proves accurate for my own personal taste profile, further models could be trained based on other people’s listening history, to test whether the basic principles remain true for all users. However, I would anticipate high variance in the performance of such models, given that different people have different musical tastes – some more heterogeneous than others.

**Evaluation Metrics**

In evaluating the performance of my final model, I will calculate the model’s accuracy, recall and precision, defined as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Recall = \frac{TP}{TP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

Where:

- TP = true positive (i.e. model correctly predicts a track is liked)
- TN = true negative (i.e. model correctly predicts a track is not liked)
- FP = false positive (i.e. model incorrectly predicts a track is liked)
- FN = false negative (i.e. model incorrectly predicts a track is not liked)

Due to the slight imbalance between classes in the dataset (45.8% ‘likes’ vs. 54.2% ‘dislikes’), using accuracy alone could overstate the performance of the model. Therefore it is important to calculate the model’s specific performance on the positive class (i.e. recall).

However, given that the objective of this project is to accurately determine whether a given track is likely to be enjoyed or not, I believe that precision is the most relevant metric to use. Hence, the final model will be optimised for precision, and will primarily be compared against the benchmark model on this basis.

If the final model performs better than the benchmark, I will consider whether it performs well enough to solve the problem as outlined in the problem statement, also taking into consideration the model’s accuracy and recall.

Should the model perform worse than the benchmark, I will reject it as a possible solution and consider whether there are any alternative approaches that may be more effective.

## II. Analysis

### Data Exploration

Table 1: Sample Data (selected columns only)

| Track | Artist | Play_count | popularity | acousticness | ... | tempo | time_signature | valence |
|---|---|---|---|---|---|---|---|---|
| 1% | Lars Frederiksen and the Bastards | 6 | 31 | 0.001 | ... | 181.567 | 4 | 0.828 |
| 10:37 | Beach House | 20 | 46 | 0.715 | ... | 96.085 | 4 | 0.482 |
| 12:51 | The Strokes | 33 | 57 | 0.312 | ... | 145.056 | 4 | 0.591 |
| 11:11 pm | The All-American Rejects | 12 | 34 | 0.001 | ... | 153.745 | 4 | 0.682 |
| 3 | Britney Spears | 1 | 54 | 0.052 | ... | 134.962 | 4 | 0.761 |

The dataset consists of 10,832 tracks and 15 potential predictor variables. Most of these variables are continuous (e.g. tempo, duration), while two are binary (explicit and mode) and two are categorical (key and time signature).

Table 1 above shows a small sample of the dataset and selected feature columns. The full list of features (and possible values) is as follows:

- **explicit**: True, False.
- **popularity**: continuous, integers only (0-100).
- **acousticness**: continuous (0-1).
- **danceability**: continuous (0-1).
- **duration_ms**: continuous (lower bound = 0).
- **energy**: continuous (0-1).
- **instrumentalness**: continuous (0-1).
- **key**: C, C#/Db, D, D#/Eb, E, F, F#/Gb, G, G#/Ab, A, A#/Bb, B.
- **liveness**: continuous (0-1).
- **loudness**: continuous (upper bound = 0).
- **mode**: Major, Minor.
- **speechiness**: continuous (0-1).
- **tempo**: continuous (lower bound = 0).
- **time_signature**: 0, 1, 3, 4, 5.
- **valence**: continuous (0-1).

Tracks are labelled according to their play count. Tracks with three plays or more are considered “liked”, while tracks with two plays or less are considered “disliked”. The distribution of these labels is as follows:

- Total number of tracks: 10,832
- Tracks with three plays or more: 4,963
- Tracks with two plays or less: 5,869
- Percentage of tracks with three plays or more: 45.82%

**Exploratory Visualisation**

Chart 1: 'popularity' Feature Distribution

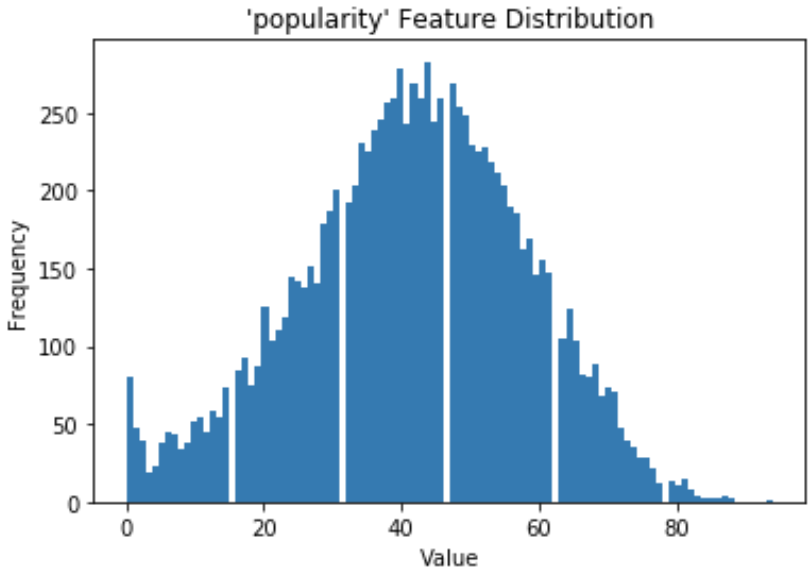



Chart 1 shows the distribution of the 'popularity' feature, which is normally distributed around its mean of 41.6. This suggests that, as calculated by Spotify [12], my listening history encompasses tracks across a range of popularities, without skewing towards either more popular or more obscure songs.

**Algorithms and Techniques**

In creating the final model, I will test three different algorithms: Decision Trees, Random Forest and XGBoost. These are all fundamentally decision tree type models, which I have previously found work well on classification problems such as these, which combine binary and continuous data inputs to solve nonlinear problems.

Decision Trees is the classic decision tree type model, which grows a single decision tree to classify each data point. It is simple and easy to implement, but prone to overfitting. Random Forest and XGBoost are ensemble methods that use specific techniques to enhance the resulting model's accuracy.

Random Forest grows multiple decision trees in parallel using random sampling of the dataset, then effectively takes the 'average' prediction of these trees. XGBoost is a gradient boosting algorithm, which grows a set of weak learners (decision trees) in sequence, 'learning' from the mistakes of the previous tree in each iteration by minimising an error function. Given that both ensemble methods are designed to improve upon the basic Decision Trees algorithm, I anticipate that both will perform better than Decision Trees on this problem.

I will first test the default implementations of these algorithms before optimising each one using a random, cross-validated parameter search [13]. This is particularly important for XGBoost, which has many tuneable parameters and is unlikely to perform well "out of the box" [14], [15]. I will use precision as the scoring metric during the optimisation process, and will select the model that performs best on this basis, while also taking recall and / or accuracy into account.

**Benchmark**

As documented in the 'Data Exploration' section above, the dataset contains 4,963 'liked' tracks – 45.8% of the total dataset. On this basis, a naïve model that classed all tracks as 'disliked' would be correct 54.2% of the time. However, this same model would have recall and precision scores of zero, illustrating its lack of any practical applications.

To create a more sophisticated benchmark model, we could consider the overall popularity of each track, as calculated by Spotify [12]. It seems reasonable to hypothesise that I might be more likely to enjoy songs with higher popularity scores, given that these songs are also enjoyed by a wider group of individuals.

Spotify's 'popularity' feature has a range of 0-100, with 100 being the most popular. If we create a model that classifies any track with a popularity score of greater than 50 as a 'like', and compare it with our earlier definition of 'likes' (i.e. at least three plays), we obtain an accuracy of 54.6% when tested on the full dataset.

While this appears to be only a slight improvement over the naïve model, this popularity-based model performs significantly better on our other evaluation metrics, as follows:

- Recall: 29.4%
- Precision: 50.8%
- F1: 37.2%

## III. Methodology

### Data Pre-processing

#### *Continuous Feature Transformation*

Chart 1 earlier demonstrated that the 'popularity' feature follows a normal distribution. Some of the other continuous features are not normally distributed however, for example 'duration_ms' and 'loudness', as charted below.

Chart 2: 'duration_ms' Feature Distribution

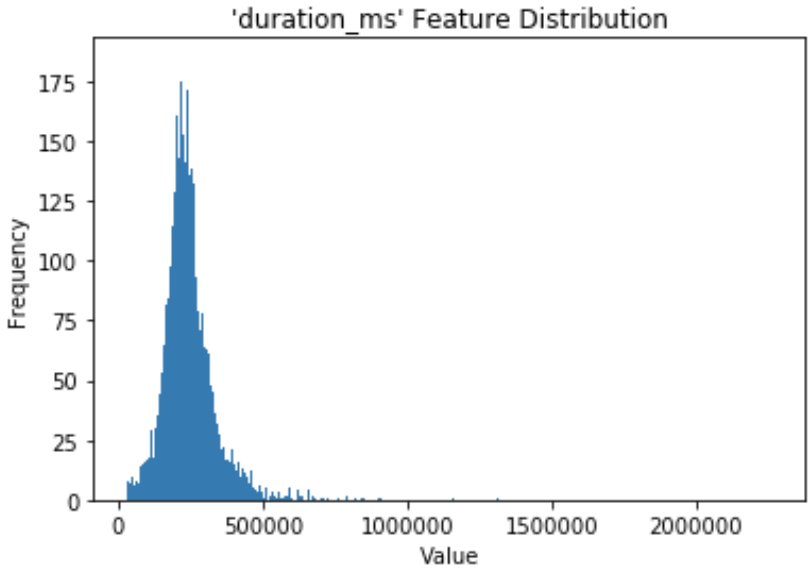
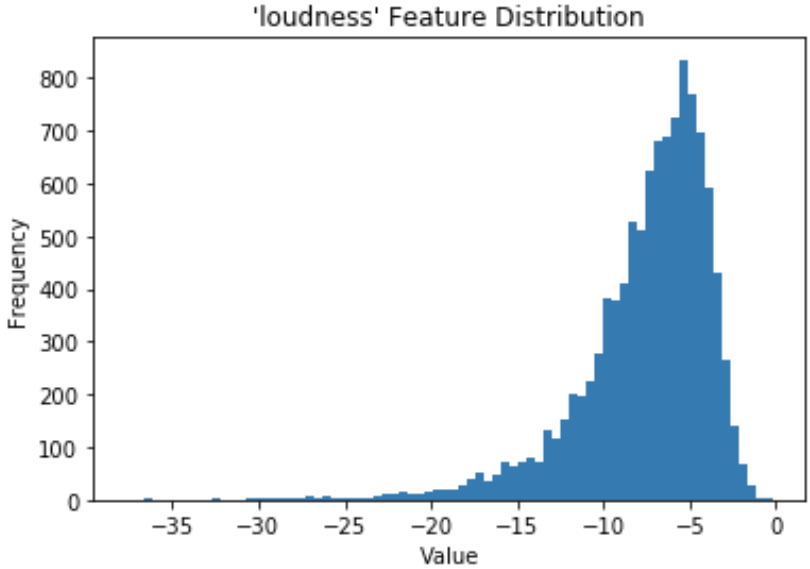



Chart 2 shows the distribution of track durations in milliseconds, which has a mean value of 246,695ms and exhibits significant positive skew. This is not surprising given the feature's lower bound of zero and unrestricted upper bound.

Chart 3: 'loudness' Feature Distribution



As shown in Chart 3, loudness (measured in decibels) has the opposite characteristic to duration, with an upper limit of zero and a (theoretically) unrestricted lower bound.

The significant skew present in both the 'duration_ms' and 'loudness' features can be reduced through logarithmic transformation [16]. This will help normalise the features' distributions, ensuring optimal algorithm performance [17]. This can be achieved using the following formulae:

- 'duration_ms': $x_{trans} = \ln(x + 1)$
- 'loudness': $x_{trans} = \ln(-x + 1)$

Where $x$ is the original value and $x_{trans}$ is the log-transformed value.

Chart 4: 'duration_ms' Feature Distribution – Log Transformed

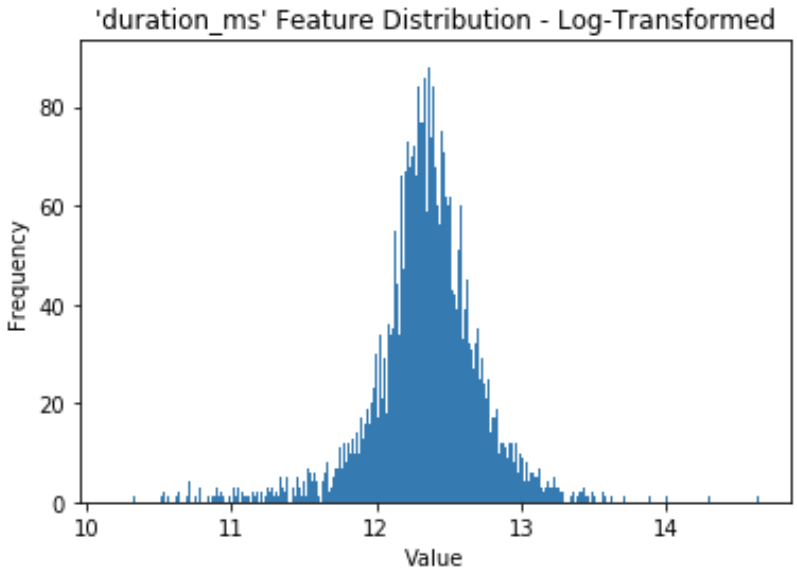
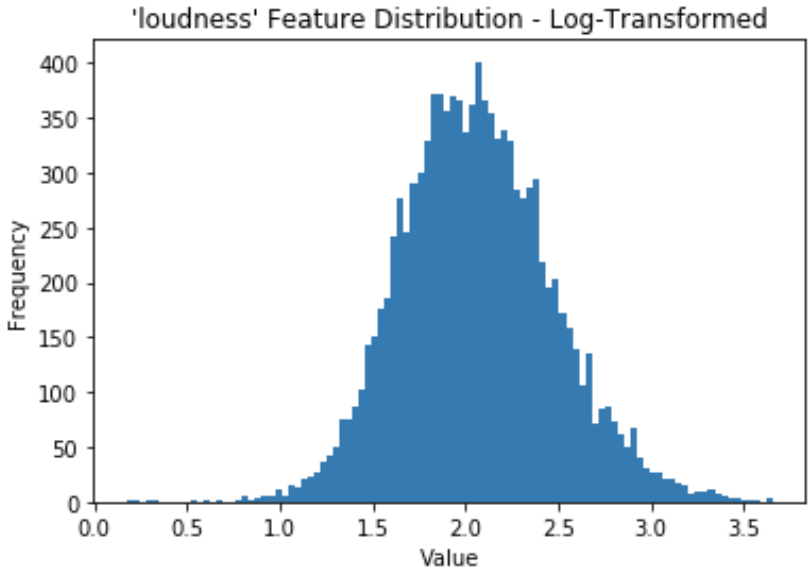



Chart 5: 'loudness' Feature Distribution – Log Transformed




Charts 4 and 5 above show the resulting 'loudness' and 'duration_ms' feature distributions after log-transformation. Both are now more normally distributed.

### *Categorical Feature Transformation*

The 'key' and 'time_signature' features are both categorical, and need to be converted to numeric (i.e. binary features). This can be achieved using 'one-hot encoding', implemented using the pandas.get_dummies function.

The 'mode' and 'explicit' features are both categorical, but binary features. Rather than use one-hot encoding, these can be converted to numeric binary features using a simple map.

After carrying out these transformations, the dataset contained 30 features in total. The final pre-processing step was to separate the data into distinct training and test sets using scikit-learn's train_test_split function, using an 80 / 20 training / test split.

## Implementation

The Decision Tree and Random Forest algorithms were implemented using the scikit-learn Python module, while XGBoost used the XGBoost module. Each classifier object was initialised using default settings and a 'random_state' of 101. The classifiers were then trained on the training data using the .fit method, and their performance was evaluated using the .predict method on the test data.

This process was extremely straightforward to carry out due to scikit-learn's consistent API, which is also compatible with XGBoost. This meant that each classifier could be trained and tested using almost identical code, the only difference occurring in the initialisation step when the specific classifier is called. For example, the Decision Trees code was as follows:

```
from sklearn.tree import DecisionTreeClassifier
clf_A = DecisionTreeClassifier(random_state = 101)
clf_A_fit = clf_A.fit(X_train, y_train)
clf_A_pred = clf_A_fit.predict(X_test)

print "Decision Trees"
print "Accuracy: " + str(accuracy_score(y_test, clf_A_pred))
print "Precision: " + str(precision_score(y_test, clf_A_pred))
print "Recall: " + str(recall_score(y_test, clf_A_pred))
print "F1: " + str(f1_score(y_test, clf_A_pred))
```

Similar code was used to train and test the Random Forest and XGBoost classifiers, using the sklearn.ensemble.RandomForestClassifier and xgboost.XGBClassifier classes.

Table 2: Default Classifier Performance on Test Data

| | Decision Trees | Random Forest | XGBoost | *Benchmark* |
|---|---|---|---|---|
| **Accuracy** | 53.7% | 58.8% | 63.2% | *54.9%* |
| **Precision** | 50.4% | 58.0% | 63.2% | *52.5%* |
| **Recall** | 49.0% | 42.6% | 50.5% | *35.9%* |
| **F1** | 50.0% | 49.1% | 56.2% | *42.6%* |

Table 2 compares the performance of the default classifiers against that of the benchmark model when tested on the test data. It can be seen that both the Random Forest and XGBoost algorithms outperformed the benchmark model in each metric, while Decision Trees had a worse accuracy and precision, but higher recall and F1 scores. The XGBoost classifier performed best overall.

**Refinement**

To ensure maximum model performance, each classifier was optimised using a random, five-fold cross-validated parameter search using precision as the scoring metric. This was implemented using scikit-learn's RandomizedSearchCV function with 50 tuning iterations and the following parameter settings (coded within Python dictionaries):

Decision Trees:

- `'max_depth': [3, None],`
- `'max_features': range(2, 31, 8) + [None],`
- `'min_samples_split': sp_randint(2, 11),`
- `'min_samples_leaf': sp_randint(1, 11),`
- `'criterion': ['gini', 'entropy']`

Random Forest:

- `'max_depth': [3, None],`
- `'max_features': range(2, 31, 8) + [None],`
- `'min_samples_split': sp_randint(2, 11),`
- `'min_samples_leaf': sp_randint(1, 11),`
- `'criterion': ['gini', 'entropy'],`
- `'n_estimators': [100]`

XGBoost:

- `'n_estimators': [100],`
- `'learning_rate': [i / 100.0 for i in range(5, 31, 1)],`
- `'min_child_weight': range(1, 13, 1),`
- `'max_depth': range(3, 7),`
- `'gamma': [i / 100.0 for i in range(0, 50)],`
- `'subsample': [i / 100.0 for i in range(50, 101)],`
- `'colsample_bytree': [i / 100.0 for i in range(50, 101)]`

Table 3: Optimised Classifier Performance on Test Data

| | Decision Trees | Random Forest | XGBoost | *Benchmark* |
|---|---|---|---|---|
| **Accuracy** | 59.7% | 60.5% | 63.6% | *54.9%* |
| **Precision** | 61.6% | 69.1% | 63.9% | *52.5%* |
| **Recall** | 36.3% | 27.7% | 50.7% | *35.9%* |
| **F1** | 45.7% | 39.5% | 56.6% | *42.6%* |

Table 3 compares the performance of the optimised classifiers on the test data against that of the benchmark model. It can be seen that the accuracy and precision of each classifier improved relative to their default implementations, but the recall (and therefore F1) scores of the Decision Trees and Random Forest classifiers fell significantly. Only the XGBoost classifier improved in every metric.

## IV. Results

### Model Evaluation and Validation

As shown in Table 3, the Random Forest and XGBoost models performed most strongly in terms of precision, with 69.1% and 63.9% respectively. However, the Random Forest model performed much worse in terms of recall, at 27.7% vs. 50.7%. While precision is the primary evaluation metric for this problem, the 23 percentage point recall performance penalty of the Random Forest model is not justified by the 5.2 percentage point increase in precision. Thus, the XGBoost model will be chosen as our final model.

The final XGBoost model had the following (non-default) parameter settings:

- **'colsample_bytree':** 0.75
- **'gamma':** 0.06
- **'learning_rate':** 0.05
- **'max_depth':** 4
- **'min_child_weight':** 2
- **'n_estimators':** 100
- **'subsample':** 0.67

These settings are consistent with a typical boosting model: a series of (100) weak learners that improve with each iteration – gradually in this case, given the relatively low learning rate.

Most of these parameters determine the construction of each individual decision tree within the ensemble. For example, the 'max_depth' setting ensures that each tree can only have up to four leaf nodes (i.e. splits), while the 'colsample_bytree' setting means that only 75% of the total columns within the data subsample will be considered for each tree split. These settings aim to reduce overfitting by controlling model complexity and adding randomness to the training process [18].

Table 4: Final Model Cross-Validation Descriptive Statistics *

| | CV Train Precision | CV Test Precision |
|---|---|---|
| **Min** | 67.7% | 58.1% |
| **Max** | 68.3% | 62.6% |
| **Mean** | 68.1% | 60.4% |
| **Std. Dev.** | 0.2% | 1.9% |

** Note in this context 'train' and 'test' refer to the training and test data within each cross-validation fold, both of which are subsets of the overall training dataset. This should not be confused with the overall test dataset, which is not seen by the model during cross-validation.*

Table 4 shows the overall results of the five-fold cross-validation process for the final model, as obtained through the random parameter search process. It can be seen that the model's performance does depend to some extent on the training data, given the 4.5 percentage point range in test precision across the five folds.

However, some variation is to be expected, and the relative consistency of the model's performance during cross-validation with its performance on the overall test dataset (63.9%) suggests that the model is sufficiently robust for its intended application.

**Justification**

Table 5: Final Model Performance on Test Data vs. Benchmark

| | Benchmark | Final Model | *Absolute Delta* | *Relative Delta* |
|---|---|---|---|---|
| **Accuracy** | 54.9% | 63.6% | *+8.7pp* | *+15.9%* |
| **Precision** | 52.5% | 63.9% | *+11.4pp* | *+21.8%* |
| **Recall** | 35.9% | 50.7% | *+14.8pp* | *+41.3%* |
| **F1** | 42.6% | 56.6% | *+13.9pp* | *+32.7%* |

Table 5 shows the performance of the final model against the benchmark. It can be seen that the final model outperformed the benchmark in every metric, including an 11.4 percentage point increase in precision score – the primary evaluation metric. On a relative basis, this is equivalent to a 21.8% performance increase.

Given the final model's considerable advantages over the benchmark, the final model will be selected to solve the problem.

While this performance comparison highlights the advantages of using a complex model over a simple one in this particular context, it is important to assess the final model on its standalone merits and consider its limitations: 63.9% is a relatively low precision score for a precision-optimised model, which highlights the inherent difficulty in trying to model musical taste – a highly subjective quantity.

Further, the on-going performance of the model, particularly with regard to songs not yet in Spotify's catalogue, will depend on the consistent performance of Spotify's musical analysis software. It is possible that an element of the model's relatively limited precision already reflects noise in the underlying data stemming from such technical inconsistencies. This is in addition to the more obvious noise stemming from inconsistencies in my musical taste and the way it has evolved over time. The selection bias inherent in the underlying dataset, as discussed earlier, is a further source of potential error and underperformance.

However, despite the final model's limitations, it still serves a useful purpose: I can be confident that I will actually enjoy at least six out of every ten tracks that it predicts I will enjoy. This provides an additional level of screening to the existing music recommendation services I use, ensuring I focus on music I am most likely to enjoy when short on time. In this sense, it can be considered an adequate solution to the problem posed at the start of the project.

## V. Conclusion

### Free-Form Visualisation

Chart 6: Normalised Weights of First Ten Most Predictive Features

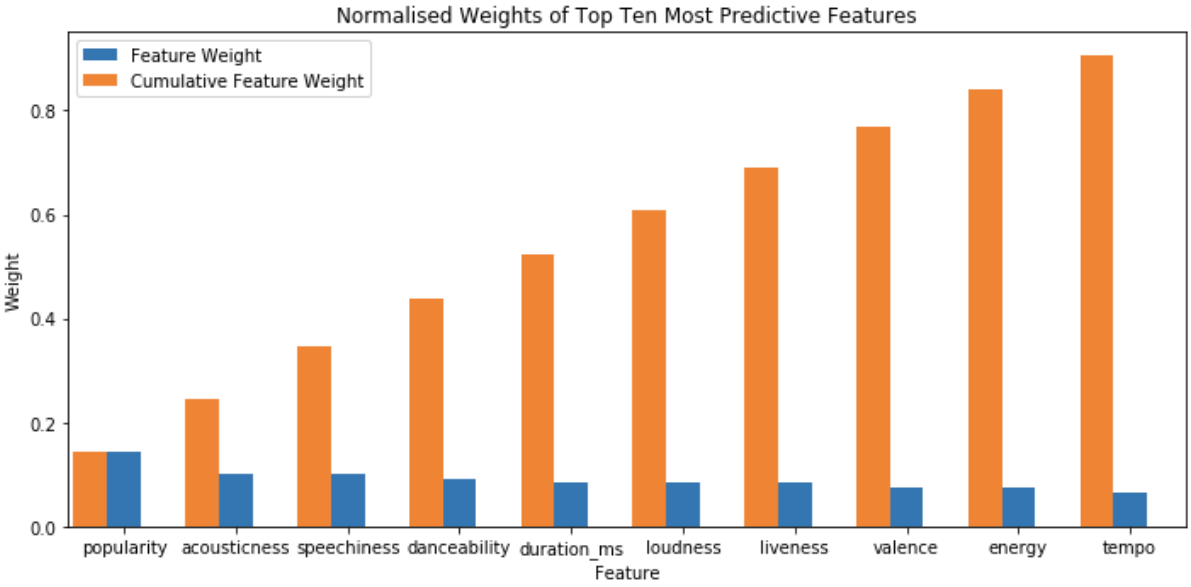



Chart 6 shows the weights of the top ten most predictive features in the default XGBoost model, defined as "the number of times a feature is used to split the data across all trees" [19]. This demonstrates that 'popularity' is the most predictive feature – 39.4% more than the second most predictive feature, 'acousticness'. The decrease in importance for each feature beyond this is less pronounced, however (e.g. 'acousticness' is only 1.5% more important than 'speechiness').

This finding suggests that 'popularity' was a reasonable feature to use in a single predictor model, but also demonstrates why that feature in isolation was limited in its predictive ability.

Not pictured in the chart are the 18 features that describe each track's key, time signature and major / minor tonality ('mode'). This is because, of the 30 total predictor variables, these features are collectively the least predictive. The most predictive of them was 'key_D#/Eb', which had a weight of 0.012 – less than a tenth that of 'popularity' (0.143).

This lack of predictive power could be due to the technical difficulty of accurately assessing musical properties like key and tonality using computer software, resulting in noisy data. It may also indicate that I don't tend to prefer songs written in any particular key. This explanation would concur with my findings from an earlier investigation, which explored the extent to which musical factors influence the performance of US number one singles [20]. I found that musical key (in isolation) was not a good predictor of the number of weeks that singles spent at number one.

### Reflection

In completing this project, I followed a step-by-step process, starting with the problem: I wanted a way to predict whether I would enjoy any particular song without listening to it, and suspected this was possible using machine learning based on the success of Spotify's Discover Weekly product.

Before working on a solution, I needed to gather the appropriate data. This was relatively easy because I track the vast majority of my listening on Last.fm, and was able to download over 11 years of playback history from there. When it came to finding musical data for each track, I was able to draw on work I'd carried out on an earlier project to obtain the data from Spotify [20].

The most difficult aspect of this project was wrangling the data effectively – specifically the process of matching songs from Last.fm in Spotify's catalogue. Lacking an intelligent means to correct errors, I had to choose whether to exclude c.300 duplicate track matches from the dataset or correct them manually. I ultimately chose the latter approach to ensure the model had as much data as possible to learn from, but this was clearly not a scalable or efficient solution. However, this would not be necessary for someone that had access to a listener's full Spotify listening history (i.e. a Spotify employee).

Once I had the clean dataset, building an effective machine learning pipeline was more straightforward. I drew on my experience from working on similar projects and established a logical flow: data exploration, feature transformation, basic model training, model optimisation, and feature importance analysis. I used a Jupyter Notebook to organise my work, which helped organise the flow.

I found the final result incredibly interesting, especially when viewed alongside the feature importances. It makes sense that track popularity would be a reasonably good predictor of enjoyment, as it effectively draws on the wisdom of the crowd. Also, at the outset I wasn't sure that it would be possible to obtain a precision of much greater than 50%, so I was pleased to be proven wrong with the final score of 63.9%. The Random Forest model showed that even greater scores are theoretically possible, albeit at the expense of completeness (i.e. recall).

To summarise, the final model and solution surpassed my (low) expectations. However, the data gathering process also highlighted the difficulty in adapting these findings to predict other peoples' musical enjoyment: most people do not track every song they listen to, making it difficult to obtain the necessary training data. However, the potential is clearly there for music streaming companies to make smart recommendations for its users, just as Spotify is already doing.

**Improvement**

The aspect of this project that most needs improving is the dataset itself: as already discussed, the potential for confirmation bias within the data is high, and the benchmark for 'like' vs. 'dislike' (three plays or more) is fairly arbitrary. A more comprehensive dataset would list every song heard in any context, ideally with a more tangible measure of enjoyment.

However, that is not practical, or likely to become practical in the foreseeable future. A more realistic improvement would be to work directly with a company like Spotify that had access to comprehensive user data, and could feasibly run experiments on song recommendations and monitor user's reactions (e.g. whether they let the track play in full or skip it). Further, a more forensic audio analysis of each track may yield a large number of new predictor variables, which could be exploited using deep learning techniques.

All of these suggestions either have been, or are presently being explored by Spotify [6]. As such, while my final model provides a useful performance benchmark, I am certain that more accurate models already exist, or are currently in development.

**References**


1. https://www.recordingconnection.com/reference-library/recording-entrepreneurs/how-the-internet-changed-music
2. https://qz.com/571007/the-magic-that-makes-spotifys-discover-weekly-playlists-so-damn-good/
3. https://link.springer.com/content/pdf/10.1007/s10994-006-9019-7.pdf
4. https://github.com/indrajithi/mgc-django
5. https://chatbotslife.com/finding-the-genre-of-a-song-with-deep-learning-da8f59a61194
6. http://benanne.github.io/2014/08/05/spotify-cnns.html
7. https://developer.spotify.com/web-api/get-audio-features/
8. https://medium.com/@polomarcus/music-recommendation-service-with-the-spotify-api-spark-mllib-and-databricks-7cde9b16d35d
9. https://medium.com/mlreview/spotify-analyzing-and-predicting-songs-58827a0fa42b
10. https://benjaminbenben.com/lastfm-to-csv/
11. https://github.com/plamere/spotipy
12. https://developer.spotify.com/web-api/get-several-tracks/
13. http://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html#sphx-glr-auto-examples-model-selection-plot-randomized-search-py
14. https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/
15. https://www.kaggle.com/tanitter/grid-search-xgboost-with-scikit-learn
16. http://oak.ucc.nau.edu/rh232/courses/eps625/handouts/data%20transformation%20handout.pdf
17. https://github.com/udacity/machine-learning/blob/master/projects/finding_donors/finding_donors.ipynb
18. http://xgboost.readthedocs.io/en/latest/how_to/param_tuning.html
19. http://xgboost.readthedocs.io/en/latest/python/python_api.html
20. https://github.com/mspbannister/dand-p4-billboard/blob/master/Billboard_analysis__100417_.md#the-billboard-hot-100-exploring-six-decades-of-number-one-singles