Bartok: Music Time Period Classification

Daniel Chiu, Derrick Liu, Yushi Wang
CS229 Final Project

Abstract

Previous work has high classification accuracy when classifying music genres that are very different [1] (e.g. rock vs. pop, jazz vs. classical), but little machine learning research has been done to classify music by subgenres within a larger genre that describe temporally and stylistically similar music. In this paper, we apply machine learning to classify classical music by time period. To do so, we used supervised learning algorithms (Naïve Bayes and SVM), unsupervised algorithms (k- nearest neighbors), and ensemble algorithms (AdaBoost) on a uniformly distributed data set, comprised of over 4,000 MIDI files extracted from an online collection. We then analyze and discuss the performance of our classifier.

Introduction

The ability to classify music has many applications, from automatic organization of musical databases to music recommendation systems. However, even to a trained human ear, differentiating between similar subgenres of music can be extremely difficult. As a result, there are rarely clear distinctions between similar genres. In particular, subgenres within music are often blurred.

Despite these difficulties however, song classification by musical style has potential. Previously, Cataltepe, Yaslan, and Sonmez were able to achieve close to 100% accuracy when classifying MIDI files into Pop, Jazz, and Classical categories [1]. In this paper, we focus on the problem of classifying music by subgroups within a specific genre in hopes of gaining insight into the limits of music classification.

Data Collection

We collected our data from a single, large source of classical music MIDI files called The Classical MIDI Connection [2]. From this source, we drew classical music from 5 time periods: Renaissance, Baroque, Classical, Romantic, and 20th Century.

The Classical MIDI Connection does not have any public API for use, so we spent a substantial amount of time writing a Python scraper to download the MIDI files for each time period. Due to the lack of structure in the website's HTML markup, we could not quickly capture these MIDI files with straightforward HTML parsing tools. Instead, we needed to apply manual parsing techniques (e.g. document tree traversal) to the webpage in order to effectively locate and download all available MIDI files.

Here is the breakdown for the number of songs we collected per time period (after filtering out bad data):

Baroque	1248
Classical	727
Renaissance	502
Romantic	1154
20th Century	974

Features

For each song, we parsed a variety of features to train classification algorithms against. To decide what features to train against, we researched the characteristics of the different time periods of classical music, noting that the key, dynamics, rhythm, instrumentation, and musical progression tend to define a period's style [5].

Parsing

We used a Python library from MIT called music21 [3]. Using this library, we were able to convert each MIDI file into a music21 stream of events. Each MIDI stream contains information about the key and time signatures for a song, as well as a list of notes and chords (in chronological order). After conversion, we used music21 modules to extract the song data and metadata.

We defined song metadata to be information contained in the song that does not require an analysis of the underlying song structure, and song data to be information specifically conferred by a song's note and chord structure. More specifically, we extracted time and key signatures, average tempo, and longest sustained tempo from each song's metadata. For the note data itself, we first used music21 to transcribe each song into its respective neutral key signature (A minor or C major) to normalize the notes, then extracted histograms the song's pitches, chord bases, and intervals between consecutive notes or chords from the song.

We created a self-contained Python object to store the features for each file. Thus, parser module outputs a single song data object per file; running the parser on each file in the collected songs list resulted in a set of objects that each correspond to a collected song.

In order to efficiently process this computationally-intensive workload, we used the `joblib` python library to effectively parallelize 8 worker threads on our set of songs. We further separate the set of songs into small chunks to reduce memory consumption per worker thread. Each worker thread would run the parser module on its subset of the data, and a final worker thread would coalesce the results at the end of parallelized computation.

Preprocessing

While each of these song objects contain useful information, the raw objects could not have been used to train our machine learning algorithm. All of the classification algorithms we utilized train classifiers on numerical vectors, so we needed to determine a scheme to compose such a vector from the various data contained in these song objects. We finalized a scheme to convert non-numerical, negative, and non-vector data, and used that scheme to translate song objects into vectors that could be directly input into our classification algorithms. Data that was already in numerical format (e.g. time signatures) were simply coalesced into a vector format. To store the more complicated features such as the note and chord histograms, we flattened each complex feature structure into a single numerical vector per feature. Finally, we concatenated many vectors to form one overall numerical feature vector per song.

Learning Algorithms and Processes

We exclusively used `scikit-learn` as the machine learning library in our project. `scikit-learn` provides extensive classification algorithm flexibility, metrics functionality, and cross-validation support [4], which proved to be valuable throughout the development and assessment of our project's performance.

Multinomial Naïve Bayes

We started out with Naïve Bayes from the `scikit-learn` module to establish baseline performance [1]. Naïve Bayes makes the assumption that features are independent, which is likely not a very good model for music. Ultimately, Naïve Bayes gave us an accuracy of just over 50%; this was far more accurate than randomly choosing a category, but it did not perform nearly as well as an SVM.

SVM

We later chose to test with SVMs because SVMs are a popular and effective algorithm and we knew that it would provide more robust classification than Naïve Bayes. We also expected SVM's to work better with our relatively large feature set. We experimented with an SVM with two different types of kernels: a linear kernel and a radial basis function (RBF) kernel [7]. In `scikit-learn`'s implementation, an RBF kernel is functionally representative of a Gaussian kernel. We found that the RBF kernel performed slightly better than the linear kernel due to the fact that the data was almost certainly not linearly separable.

Later on, we found that SVMs worked better as a whole when we switched from an SVM that was solving the dual problem to an SVM solving the primal problem.

K-nearest Neighbors

For variety, we also tried a k-nearest neighbors classifier on our data set to compare the performance of unsupervised learning with more traditional supervised learning algorithms. We found that the classifier worked best when we set it to look at the 6-nearest neighbors, but that it did not perform as well as an SVM, especially when the SVM was initialized with tuned parameters and kernels.

AdaBoost

After using the four algorithms mentioned, we decided to try running AdaBoost, short for Adaptive Boosting. AdaBoost is a weighted ensemble algorithm that repeatedly fit a sequence of weak learners to our data [6]. The rationale behind this was that such a boosting algorithm would naturally be very flexible, due to the large number of classifiers hidden behind the algorithm. Furthermore, it's very resistant to overfitting, and requires no knowledge of the weak learners while providing decent theoretical guarantees on accuracy [6].

Cross-validation processes

We trained these classification algorithms on stratified 5-folds of preprocessed data vectors. We chose to use stratified K-folding to minimize overfitting to a training set, while maintaining that the proportion of data represented in each fold is the same as the proportions of actual data.

Design Decisions

We initially decided to utilize a multinomial Naïve Bayes classifier to provide us with general insight into the strengths and weaknesses of our data set, feature selection, and implementation. We reasoned that multinomial Naïve Bayes is simple, straightforward, and easy to use, starting with it would allow us to quickly iterate and improve our project.

In this initial multinomial Naïve Bayes round, we trained the classifier with 480 examples and three initial features (note histogram, key signature, time signature), and achieved an average accuracy of 40% over 5 folds of 96 examples each.

Fold number	*Accuracy*	*mislabel / total*
Fold 1	0.406250	57 / 96
Fold 2	0.364583	61 / 96
Fold 3	0.395833	58 / 96
Fold 4	0.416666	56 / 96
Fold 5	0.416666	56 / 96

We also generated and plotted training-test accuracy, which revealed that multinomial Naïve Bayes was experiencing both high variance and high bias. This prompted us to expand our number of training examples, and the feature set that Naïve Bayes was being trained on.

To address the lack of training examples, we expanded our data collection parameters to include composer-specific pieces provided on The Classical MIDI Connection's website. Including these composer-specific pieces quadrupled our training set size – from 1248 to 4605 examples. Training the same Naïve Bayes classifier with these new data practically eliminated the high variance experienced earlier when plotting training-test errors.

Originally, we had six categories: Baroque, Classical, Renaissance, Romantic, Impressionistic, and 20th century. We noticed that our classifiers were getting confused between Impressionistic and 20th century more so than between other time periods. To resolve this, we decided to fold these two categories together since the Impressionistic movement occurred during the 20th century, and can be considered a subset of contemporary music. After making this change our accuracy increased from 66.98% to 71.04%.

Results

Our final results were better than we expected. If we were to randomly choose between categories we would expect 20% accuracy. Using our best classifier, we were able to achieve 71.04% accuracy. This is a drop of 9% compared to the 80% accuracy for identifying pop from jazz from classical in exchange for having songs that are far more similar to each other.

Adding in all of our new features and new data, the performance of our initial Naïve Bayes increased from 40% to 56%. This was the least effective of the five things we tried.

SVM performed even better, achieving 70.48% accuracy with a linear kernel and 71.04% accuracy with an RBF kernel. For the linear kernel, we found that we achieved about 5% higher accuracy when we switched from a kernel that solved the dual problem to one that solved the primal problem. Since 71.04% accuracy was the highest accuracy we found out of all the algorithms we tried, we have also provided some graphs visualizing the RBF SVM confusion matrix and training vs. test error (see Figure 2). From this matrix, we note that confusion between time periods often occurs when the periods are adjacent to one another. For example, in the above figure, Baroque pieces are often confused with Renaissance, since Baroque pieces likely draw influences from pieces written during the Renaissance period immediately preceding.

Naïve Bayes performed the worst among our classifiers. This is probably due to the complexity of the features and the simplicity of the algorithm itself, which allows for speed, but also larger errors.

Overall, for supervised algorithms, SVMs performed the best among all the trained classifiers by a significant margin, with the nonlinear (RBF) SVM slightly edging out the linear SVM, possibly due to common musical influences, as mentioned above.

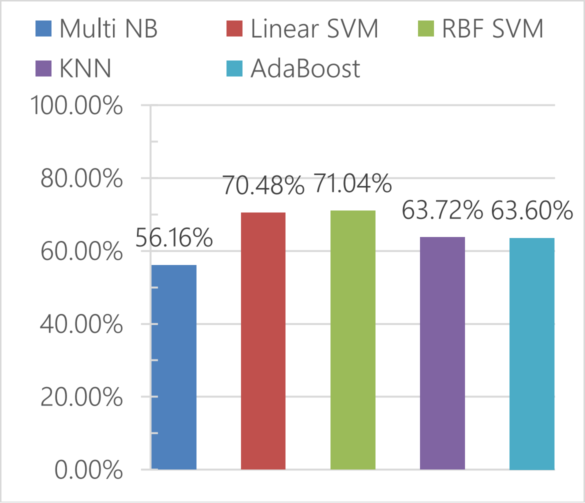

Figure 1. Cumulative overview of final music classification mean accuracies

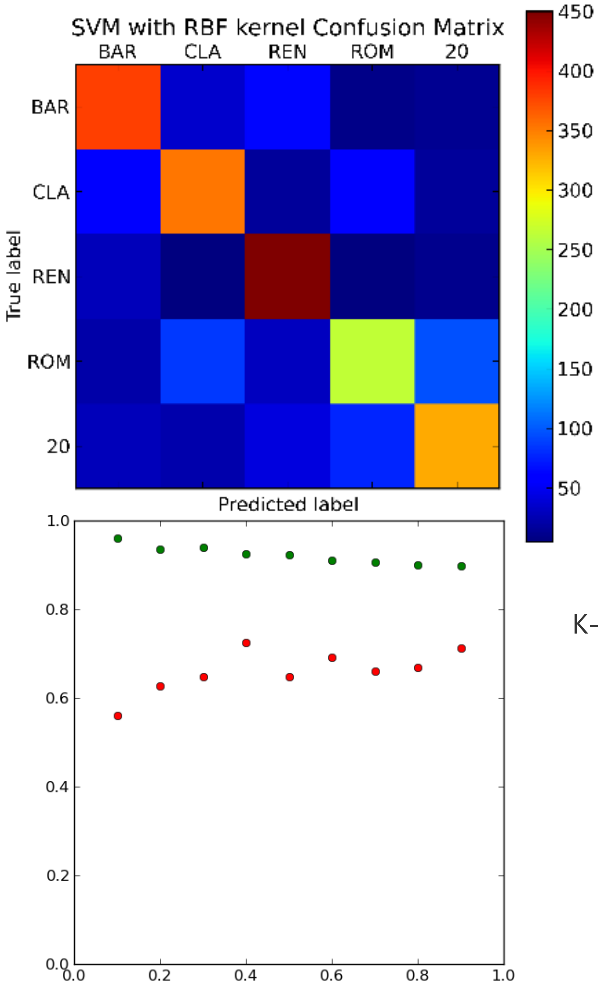

Figure 2. Confusion matrix and training-test accuracy plot from SVM classification, with RBF kernel

K-nearest neighbors supports two separate weighting modes: distance, in which the weighted influence of a point is a function of its distance, and uniform, in which all points are weighted equally regardless of distance. We found that using the distance weighting mode worked best. Additionally, k-nearest neighbors supports several different distance metrics including Manhattan (l_1) distance and Euclidean (l_2) distance. Manhattan (l_1) distance yielded the best results. We also found that the algorithm produced the highest accuracy when we set $k = 6$. With these parameters, our k-nearest neighbors classifier was able to achieve 63.44% accuracy.

Finally, AdaBoost achieved a somewhat disappointing 63.6% accuracy. We found that we had the highest performance when we used 150 estimators with a learning rate of 0.1. Since two songs from different time periods might be more similar than two songs from two different time periods, significant noise could be present in the training data. Boosting algorithms like AdaBoost are susceptible to noise, which might explain AdaBoost's poor performance [6].

Conclusions and future work

Although the time periods in classical music are very similar, machine learning was able to provide relatively accurate period classification. We observed how SVMs, with their propensity for high-dimensional features, were able to dominate other classifiers. Meanwhile, we also observed how AdaBoost, while very useful in other fields, was seemingly unsuited for music classification.

Further research could improve Bartok, and might include:

- Using Principal Component Analysis to reduce feature noise and bloat
- More complex features (e.g. chord progression) to potentially improve accuracy
- Expanding input to also include audio files and other types of musical notation
- Application to music domains outside of classical music to tackle different genres or any other form of music

Acknowledgements

First and foremost, we would like to thank Andrew Ng for his invaluable lectures and machine learning advice, and the entire CS 229 staff for their support throughout this quarter.

We would also like to thank Windows Azure, which provided computational credit for our project. We decided to use Azure because it provided dedicated high performance computation and a central platform for project development.

References

[1] McKay, Cory and Fujinaga, Ichiro, "Automatic genre classification using large high-level musical features," in Proceedings of the International Conference on Music Information Retrieval, 2004, pp. 525-530. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.2530

[2] "Classical MIDI Connection." Classicial Midi Connction. Accessed 01 Oct. 2013. http://www.classicalmidiconnection.com/classical.html.

[3] "Music21: A Toolkit for Computer-aided Musicology." music21. Aug. 2013. Accessed 05 Oct. 2013. http://web.mit.edu/music21/.

[4] Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.

[5] Langlois, Jeffrey B. "Musical Styles". Music Anthology. 10 June, 2008. Accessed 05 Oct. 2013. http://www.musicanthology.org/?p=50

[6] Freund, Yoav; Schapire, Robert E. (1995). *A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting*. CiteSeerX: 10.1.1.56.9855

[7] Andreas Müller (2012). Kernel Approximations for Efficient SVMs (and other feature extraction methods)