Bach Chorales Classification And Chord Prediction

Shuchang Liu
PID: A53276224
UCSD Electrical and Computer Engineering
s7liu@ucsd.edu

Arik Horodniceanu
PID: A53285765
UCSD Electrical and Computer Engineering
ahorodni@ucsd.edu

***Abstract*— This paper presented an thorough analysis on Bach Chorale Harmony Dataset, which contains pitch class information from MIDI data of Bach chorales. Based on the information provided, we will use multiple supervised learning methods to classify chorales and predict chords in a certain chorale event. The result shows that this was a nice try.**

I. INTRODUCTION

Chorale is the name of several related musical forms originating in the music genre of the Lutheran chorale. The chorale originated when Martin Luther forked sacred songs in vernacular language from established practices of church music near the end of the first quarter of the 16th century. The first hymnals according to Luther's new practice were published in 1524.

After the introduction in Lutheran churches, in the early 18th century, the format was soon expanded with choral movements in the form of four-part chorales. Composers such as Johann Sebastian Bach and Gottfried Heinrich Stölzel placed these chorales often as a concluding movement at the end of their church compositions.

In this project, we will use supervised learning methods such as Naive Gaussian Bayes, SVM, Random Forest, etc to classify chorales and predict the chord played in certain chorale event. The rest of paper is arranged as follows. First we give a brief introduction about the dataset we use in Section II. The we will describe data preprocessing and feature construction steps in Section III. Technical approaches are introduced in Section IV. At last we setup the experiment, results are presented in Section V.

II. DATASET DESCRIPTION

The dataset we use is Bach Chorale Harmony Data. The dataset contains information of 60 chorales which are divided into 5665 events, each event was described by event number, occurrence of 12 pitches (12 features), bass, meter and chord label.

In this paper, we shuffled the dataset and then divided it into training dataset and testing dataset by 7:3, The training dataset has 3966 events and testing dataset has 1699 events.

III. DATASET PREPROCESSING

A. Data Cleaning and Feature Construction

1) Merge same basses: According to the original dataset, there are 16 kinds of basses appeared in the dataset: A, A#, Ab, B, Bb, C, C#, D, D#, Db, E, Eb, F, F#, G, G#. However, as shown in Figure 1, C#, D#, F#, G#, A# are the same as Db, Eb, Gb, Ab, Bb, so we merged the records of C# and Db, D# and Eb, etc by transforming them into 'Xb' form. Finally, we get 12 valid basses: A, Ab, B, Bb, C, D, Db, E, Eb, F, G, Gb.

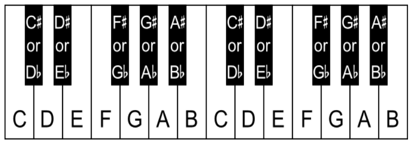

Fig. 1: Pitch classes

2) Label encoding: We converted the YES/NO label in 12 pitch features into 1/0, we encoded the long chorale ID and chord name by using 0-59 and 0-101 int number arrays.

3) One-hot encoding: According to the dataset, There are 5 kinds of meters (1,2,3,4,5) 12 kinds of basses (0-12) and 102 kinds of chords (0-101), we use one-hot encoding to convert class labels into vectors, so that the distance between every class is the same.

4) Choose the chorales and chords we are going to analyze: From Figure 2 we can see that the distribution of chorales is overall balanced, we needn't consider the unbalanced label problem during training process. However, from Figure 3 we find that the distribution of chords is highly unbalanced, so we only pick up 16 most frequent chords which have been appeared in the dataset for more than 125 times.

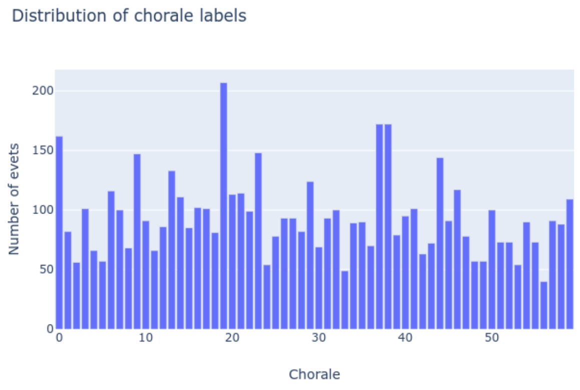

Fig. 2: Number of events in each chorale

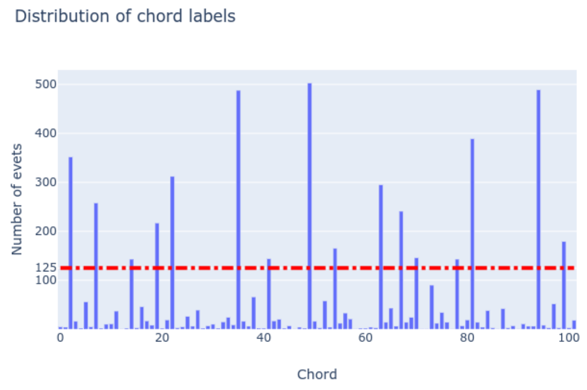

Fig. 3: Number of events for each chord

As for the rest 86 chords appeared fewer than 125 times, we will merge them into a new class so that there are 17 classes to predict in chords prediction (16 most frequent chords and 1 rare chords class).We think this is a reasonable and necessary step because some chords only appear 2 or 3 times, the dataset couldn't provide enough information for these chords. In summary, we will classify all the chorales in chorale classification and predict only the most frequent chords in chords prediction.

B. *Feature Selection*

In the chorale classification, we will use 12 pitch features, bass, meter and chord label to classify chorales. In the chord prediction, we will use 12 pitch features, bass and meter to predict chords. Event number is not used in both tasks.

IV. Technical Approach

In this section we will discuss algorithms and models used in this project.

A. *Naive Gaussian Bayes*

In machine learning we are often interested in selecting the best hypothesis h given data d.

In a classification problem, our hypothesis h may be the class to assign for a new data instance d.

One of the easiest ways of selecting the most probable hypothesis given the data that we have that we can use as our prior knowledge about the problem. Bayes' Theorem provides a way that we can calculate the probability of a hypothesis given our prior knowledge.

Bayes' Theorem is stated as: $P(h|d) = \frac{P(d|h)*P(h)}{P(d)}$. Where:

$P(h|d)$ is the probability of hypothesis h given the data d. This is called the posterior probability. $P(d|h)$ is the probability of data d given that the hypothesis h was true. $P(h)$ is the probability of hypothesis h being true (regardless of the data). This is called the prior probability of h. $P(d)$ is the probability of the data (regardless of the hypothesis). In Gaussian Naive Bayes, we assume the Gaussian (or Normal distribution) form of the probability distributions as it is the easiest to work with because we only need to estimate the mean and the standard deviation from the training data.

B. *Support Vector Machine*

For a decision hyper-plane $\mathbf{x}^T\mathbf{w} + b = 0$ to separate the two classes $P = \{(\mathbf{x}_i, 1)\}$ and $N = \{(\mathbf{x}_i, -1)\}$, it has to satisfy

$$y_i(\mathbf{x}_i^T\mathbf{w} + b) \geq 0$$

for both $\mathbf{x}_i \in P$ and $\mathbf{x}_i \in N$. Among all such planes satisfying this condition, we want to find the optimal one H_0 that separates the two classes with the maximal margin (the distance between the decision plane and the closest sample points).

The optimal plane should be in the middle of the two classes, so that the distance from the plane to the closest point on either side is the same. We define two additional planes H_+ and H_- that are parallel to H_0 and go through the point closest to the plane on either side:

$$\mathbf{x}^T\mathbf{w} + b = 1, \quad \text{and} \quad \mathbf{x}^T\mathbf{w} + b = -1$$

All points $\mathbf{x}_i \in P$ on the positive side should satisfy

$$\mathbf{x}_i^T\mathbf{w} + b \geq 1, \quad y_i = 1$$

and all points $\mathbf{x}_i \in N$ on the negative side should satisfy

$$\mathbf{x}_i^T\mathbf{w} + b \leq -1, \quad y_i = -1$$

These can be combined into one inequality:

$$y_i(\mathbf{x}_i^T\mathbf{w} + b) \geq 1, \quad (i = 1, \cdots, m)$$

The equality holds for those points on the planes H_+ or H_-. Such points are called *support vectors*, for which

$$\mathbf{x}_i^T\mathbf{w} + b = y_i$$

i.e., the following holds for all support vectors:

$$b = y_i - \mathbf{x}_i^T\mathbf{w} = y_i - \sum_{j=1}^{m} \alpha_j y_j (\mathbf{x}_i^T\mathbf{x}_j)$$

When the two classes are not linearly separable (e.g., due to noise), the condition for the optimal hyper-plane can be relaxed by including an extra term:

$$y_i(\mathbf{x}_i^T\mathbf{w} + b) \geq 1 - \xi_i, \quad (i = 1, \cdots, m)$$

For minimum error, $\xi_i \geq 0$ should be minimized as well as $||\mathbf{w}||$, and the objective function becomes:

$$\begin{aligned} &\text{minimize} \quad \mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\xi_i^k \\ &\text{subject to} \quad y_i(\mathbf{x}_i^T\mathbf{w} + b) \geq 1 - \xi_i, \quad \text{and} \quad \xi_i \geq 0; \quad (i = 1, \cdots, m) \end{aligned}$$

Here C is a regularization parameter that controls the trade-off between maximizing the margin and minimizing the training error. Small C tends to emphasize the margin while ignoring the outliers in the training data, while large C may tend to overfit the training data.

C. Random Forest

Random forests is an ensemble learning algorithm. The basic premise of the algorithm is that building a small decision-tree with few features is a computationally cheap process. If we can build many small, weak decision trees in parallel, we can then combine the trees to form a single, strong learner by averaging or taking the majority vote. In practice, random forests are often found to be the most accurate learning algorithms to date. The pseudocode is illustrated in Algorithm 1. The algorithm works as follows: for each tree in the forest, we select a bootstrap sample from S where S (i) denotes the ith bootstrap. We then learn a decision-tree using a modified decision-tree learning algorithm. The algorithm is modified as follows: at each node of the tree, instead of examining all possible feature-splits, we randomly select some subset of the features $f \subseteq F$. where F is the set of features. The node then splits on the best feature in f rather than F. In practice f is much, much smaller than F. Deciding on which feature to split is oftentimes the most computationally expensive aspect of decision tree learning. By narrowing the set of features, we speed up the learning of the tree.

D. XGBoost

Boosting methods build models from individual so called "weak learners" in an iterative way. In boosting, the individual models are not built on completely random subsets of data and features but sequentially by putting more weight on instances with wrong predictions and high errors. The general idea behind this is that instances, which are hard to predict correctly ("difficult" cases) will be focused on during learning, so that the model learns from past mistakes. When we train each ensemble on a subset of the training set, we also call this Stochastic Gradient Boosting, which can help improve generalizability of our model.

The gradient is used to minimize a loss function, similar to how Neural networks utilize gradient descent to optimize ("learn") weights. In each round of training, the weak learner is built and its predictions are compared to the correct outcome that we expect. The distance between prediction and truth represents the error rate of our model. These errors can now be used to calculate the gradient. The gradient is nothing fancy, it is basically the partial derivative of our loss function – so it describes the steepness of our error function. The gradient can be used to find the direction in which to change the model parameters in order to (maximally) reduce the error in the next round of training by "descending the gradient".

In Neural nets, gradient descent is used to look for the minimum of the loss function, i.e. learning the model parameters (e.g. weights) for which the prediction error is lowest in a single model. In Gradient Boosting we are combining the predictions of multiple models, so we are not optimizing the model parameters directly but the boosted model predictions. Therefore, the gradients will be added to the running training process by fitting the next tree also to these values.

XGBoost stands for "Extreme Gradient Boosting"; it is a specific implementation of the Gradient Boosting method which uses more accurate approximations to find the best tree model. It employs a number of nifty tricks that make it exceptionally successful, particularly with structured data. The most important of which are:

1.) computing second-order gradients, i.e. second partial derivatives of the loss function (similar to Newton's method), which provides more information about the direction of gradients and how to get to the minimum of our loss function. While regular gradient boosting uses the loss function of our base model (e.g. decision tree) as a proxy for minimizing the error of the overall model, XGBoost uses the 2nd order derivative as an approximation.

2.) And advanced regularization (L1 L2), which improves model generalization.

XGBoost has additional advantages: training is very fast and can be parallelized / distributed across clusters.

E. Neural Networks

Here we use a Neural network of being a multilayer perceptron. Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot) : R^m \rightarrow R^n$ by training on a dataset, where m is the number of dimensions for input and 0 is the number of dimensions for output. Given a set of features X and a target y, it can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. The first layer, known as the input layer, consists of a set of neurons representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation $\sum_i w_i x_i$, followed by a non-linear activation function $g(\cdot) : R \rightarrow R$ - like the hyperbolic tan function. The output layer receives the values from the last hidden layer and transforms them into output values. The leftmost layer, known as the input layer, consists of a set of neurons representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation , followed by a non-linear activation function - like the hyperbolic tan function. The output layer receives the values from the last hidden layer and transforms them into output values. The advantages of Multi-layer Perceptron are:

1) Capability to learn non-linear models.

2) Capability to learn models in real-time (on-line learning) using partial$_fit$.

The disadvantages of Multi-layer Perceptron (MLP) include:

1) MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.

2) MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations.

3) MLP is sensitive to feature scaling.

V. Results

We used the techniques detailed in section IV to classify chords and chorales. The results are given in percentages out of 100%.

A. Chorale Classification

Our results shows that the methods used worked very poorly, both in training and during test times.

TABLE I: Model performance in different models for chorle classification

	Train Accuracy	Test Accuracy
Gaussian Naive Bayes	11.35	9.13
SVM	52.9	16.07
Random Forest	53.13	18.95
XGboost	53.03	19.66
Neural Network	53.13	18.83

B. Chord Classification

Our results shows that the methods used worked better for chord classification

TABLE II: Model performance in different models for chord classification

	Train Accuracy	Test Accuracy
Gaussian Naive Bayes	11.27	8.06
SVM	83.56	66.98
Random Forest	87.67	74.57
XGboost	87.37	73.63
Neural Network	87.67	72.75

From the results, we observed that predicting chords can reach higher accuracy than classifying chorales, except for Gaussian Naive Bayes. We think this is because the model is too simplistic, even for chord classification.

As for the reason, we guess may be it's because the difference in feature space between different chorales is not as significant as the difference between different chords. Take the 12 pitch features as an example, as shown in Figure 4 and Figure 5, the different between different chorales is hard to find. For example, we can see that A pitch appears in all chorales are at almost the same frequency while A pitch frequency is significantly different in different chords.

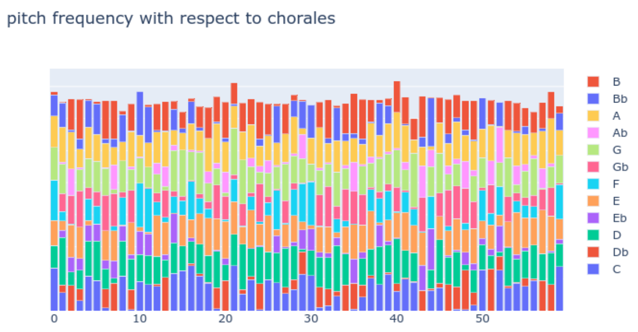

Fig. 4: Pitch frequency with respect to chorales

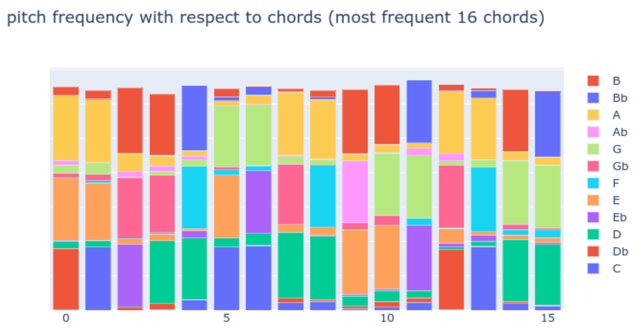

Fig. 5: Pitch frequency with respect to chords

VI. Conclusion

In all, we see that chorale classification worked very poorly, while we are able to classify chords to some accuracy. As explained previously, we think this is because we can find that the pitch features are easier to capture when considering chords than when considering chorales.

References

[1] https://en.wikipedia.org/wiki/Chorale
[2] https://en.wikipedia.org/wiki/Chord
[3] Nir Friedman, Dan Geiger, and Moises Goldszmidt. 1997. Bayesian Network Classifiers. Mach. Learn. 29, 2-3 (November 1997), 131-163. DOI: https://doi.org/10.1023/A:1007465528199
[4] Marti A. Hearst. 1998. Support Vector Machines. IEEE Intelligent Systems 13, 4 (July 1998), 18-28. DOI: https://doi.org/10.1109/5254.708428
[5] Leo Breiman. 2001. Random Forests. Mach. Learn. 45, 1 (October 2001), 5-32. DOI: https://doi.org/10.1023/A:1010933404324
[6] Chen, Tianqi, and Carlos Guestrin. "XGBoost." Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16 (2016): n. pag. Crossref. Web.
[7] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation applied to handwritten zip code recognition. Neural Comput. 1, 4 (December 1989), 541-551. DOI=http://dx.doi.org/10.1162/neco.1989.1.4.541