Machine Learning Final Project

Wei-Tang Chien, Tien-Leng Wu, Guan-Ming Su

Abstract

It is our honor to have this opportunity to choose "danceable" tracks for the upcoming party. We try more than three different models and three of them are chosen as the "best" ones for this report: **Boosting Methods Combined with Residual Learning, Deep Learning, and Ranking Regression Based on Binary Classification**. For each model, a brief overview about the model structure and its characteristics will be given, followed by the performance assessment. Finally, we will recommend Nijika Chan which model to use, and provide her with pros and cons.

1 Data Preprocessing

i) We abandon some features such as "id" and features related to "url" due to the fact that they are considered useless in our further analysis.

ii) We observe that there are plenty of null values (`NaN`) in our data, which have to be taken care of carefully before training. The null values appear in all kinds of features, such as numerical features, boolean features, and string features, and there is no doubt that we need to treat them differently:

- If a null value is numerical, the median of the corresponding feature will be filled.
- If a null value is of boolean type, we fill the majority of that feature into it.
- If a null value is a string, we simply fill it with "`--`".

iii) We then convert `False`/`True` (boolean value) to `0`/`1`.

iv) Since the string values cannot be directly handled by machine, we perform the following 3 further preprocessing on string features, according to their properties.

- For features "album type", "composer" and "artist," we select **One-Hot-Encoding** as the string-to-numeric feature conversion, simply because after checking the unique values by numpy.unique function, we realize that there are only 4, 11, 97 different values for "album type", "composer" and "artist," respectively, among the training set. Unique values for "album type" and "composer" in the testing set are identical to that in the training set; there are only 96 unique values for "artist" in the testing set, but fortunately them are all in the training set unique values. These unique values having small-size (below 100) motivate the One-Hot Encoding since we can represent the corresponding features with One-Hot vectors having reasonable length.
- For features "channel" and "title", we employ **Tokenization**, which is a popular way for text processing, to convert them into numeric values because after checking their number of unique values, we figure out that there are numerous unique values for them, which means that One-Hot Encoding is definitely not the best choice for processing these features. Another thing is that the feature "description" is also tokenized but jettisoned in the end since its dimension is still too large after tokenization.
- For features "album" and "track", if we try to encode these features to One-Hot vector, the vector will be too long because there are too many distinct albums or tracks. Therefore, we compute the average danceability for each albums and tracks that have appeared in the training set. We then convert the values in those two features in the testing set based on the statistical information just derived from the training set; as for albums and tracks which do not appear in the training set, we simply give 0 to their values. Our preprocessing method may be rational because when given an unknown song, we can only guess its danceability by the album and the track it belongs to. This is quite similar to human's mindset: "Guessing by its own characteristics."

Table 1 summarizes the preprocessing procedures, and the total number of features **before/after** the preprocessing.

Type of Data	numerical	boolean	string features whose unique values below 100	string features whose unique values above 100	
Number of Features **before** Preprocessing	14	2	3	5	
`NaN` Filled	median	majority	“`--`”	“`--`”	
Further Processing	no	0/1 conversion	One-Hot-Encoding	Tokenization	self-encoding
Number of Features **after** Preprocessing	14	2	4 (“album type”) 11 (“composer”) 97 (“artist”)	511 (“description”) 28 (“channel”) 54 (“title”)	1 (“track”) 1 (“album”)

Table 1: Summary for the Preprocessing Performed on Each Kind of Features, and How Number of Features Changes after Preprocessing

2 Models

2.1 Boosting Methods Combined with Residual Learning

2.1.1 Further Data/Label Processing

i) Include the extra dataset `test_partial_answer.csv` given in the 2nd stage of competition as “extra training data.”

ii) Shift the non-string features, i.e., features with boolean, numerical and integer values, by their corresponding mean values; then, rescale the shifted values by the standard deviation of each feature. For the tokenized-string features (notice that we only include “channel” and “title” here), after all the words are converted into numbers by the tokenizer, we further subtract them by the mean value of **all numbers**, and do rescaling using the standard deviation of **all numbers** as well.

iii) Concatenate the non-string, tokenized, and One-Hot encoded features together into a single unified training data set. Then randomly split it into the training and the validation set by train-valid-ratio $0.9 : 0.1$.

iv) As we shall see, since regressor is utilized for this model, we manually set those predicted “Danceability” greater than 9 to 9, and those less than 0 to 0; we further apply a digitizer which performs flooring on all the predictions made during/after the training. We choose **flooring instead of rounding** because of better performance on Kaggle public score.

2.1.2 Methods

The learning algorithm proposed here consists of two learning stages:

i) Train a weak model (hereinafter called **preliminary model**) with acceptable predictive ability. Here we use the simple linear regression with reinforcement by the gradient boosting algorithm, following the algorithm depicted in pseudo-code 1: In real-world implementation, we implement the “early stop”: if the mean validation error

Algorithm 1 Gradient Boosting Algorithm for Linear Regression

```
1: $s_1, s_2, \ldots, s_N = 0$
2: for each $t \in [1, T]$ do
3:     Optimize linear regression model $g(t)$ on the training data set $\mathcal{D}(\{\mathbf{x}_n, y_n - s_n\})$
4:     Obtain the coefficient $\alpha_t$ by: $\frac{\sum_{n=1}^{N} g_t(\mathbf{x}_n)(y_n - s_n)}{\sum_{n=1}^{N} [g_t(\mathbf{x}_n)]^2}$
5:     $s_n \leftarrow s_n + \alpha_t g_t(\mathbf{x}_n)$
6: end for
7: Return prediction $G(\mathbf{x}_n)$ as $\sum_{n=1}^{T} \alpha_t g_t(\mathbf{x}_n)$
```

$\langle E_{\text{valid}} \rangle$ (computed for the latest 100 iterations) does not drop below $\langle E_{\text{valid}} \rangle_{\min} + \epsilon$ within 10 early-stop-checks, the for loop will break; the early-stop-check is performed every 100 boosting iterations. Here $\langle E_{\text{valid}} \rangle_{\min}$ is the current minimum of the mean validation error, and ϵ is some tolerance, set as a small value 0.001. We add an extra preprocessing procedure for this preliminary model by transforming all the features with sklearn.decomposition.PCA method.

ii) Train another model (hereinafter called **residual estimator**), which is specialized for predicting the residual between the ground truth and the prediction by model i). We utilize the GradientBoostingRegressor in scikit-learn package, which is a regression tree based gradient boosting algorithm. The optimization process is demonstrated by the pseudo-code 2:

Algorithm 2 Gradient Boosting Algorithm for Regression Tree

```
Hypothesis with error function $\mathrm{err}(s, y)$ for each data point
1: $s_1, s_2, \ldots, s_N = 0$
2: for each $t \in [1, T]$ do
3:     Optimize linear regression tree $g(t)$ on the training data set $\mathcal{D}\left\{\mathbf{x}_n, \frac{\partial \mathrm{err}}{\partial s}\Big|_{s=s_n}\right\}$
4:     Obtain the coefficient $\alpha_t$ by minimizing: $\frac{1}{N}\sum_{n=1}^{N} \mathrm{err}\left((y_n - s_n), \alpha_t g_t(\mathbf{x}_n)\right)$
5:     $s_n \leftarrow s_n + \alpha_t g_t(\mathbf{x}_n)$
6: end for
7: Return prediction $G(\mathbf{x}_n)$ as $\sum_{n=1}^{T} \alpha_t g_t(\mathbf{x}_n)$
```

We set the maximum depth for the tree to be 8, learning rate to be 0.005, `min_samples_split` to be 2, and choose MAE as the loss function empirically based on Kaggle public score; the maximum number of boosting T is determined by **the validation MAE for the combing models, i.e., after adding the predicted residual to the prediction of the preliminary model**. Notice that the prediction made by the preliminary model is added into the original dataset as an extra feature while training this residual estimator.

By combining model i) and model ii), i.e., adding the residual to the rough guess, we anticipate that a more proper estimation of true "Danceability" can be given.

2.1.3 Model Characteristics

i) **Efficiency:**

- Need to train two models **sequentially** (though training the first one runs relatively fast with early stop) since the input and labels for residual estimator depends on the prediction of preliminary model.
- The total training time for searching the optimal number of boosting iterations (from 750 to 1200 with interval 50) for the residual estimator is roughly 3000 seconds, extremely time-consuming.
- Naïve usage of GradientBoostingRegressor is executed by single processor, i.e., failing to leverage all the computational resource.

ii) **Scalability:**
The most time-consuming part for this model is the training of the regression tree algorithm, whose time complexity (for **training a single tree**) is : $O(D \cdot N \log(N)) + O(D \cdot N) \approx O(D \cdot N \log(N))$ (see reference here), where D is the feature dimension and N is the number of training samples. By calculating the time spent on the optimal T searching process for $N = 17801 \cdot 0.9 \approx 16021$ and $N = 18709$ (increase the weighting of extra training dataset by duplication) cases respectively, we find the computation time is consistent with the formula, growing from 3240 sec to 3832 sec. Though the growth rate can be treated as linear for both N and D, given that running the GradientBoostingRegressor for $T \approx O(10^2 - 10^3)$ once is already quite expensive, we think that it will be more efficient to train different T by different processors in a parallel manner if possible.

iii) **Instability:**
The model suffers from a fluctuation in MAE (ranging from 1.74 to 1.78) among different Kaggle submissions, depending on the random seed for training-validation splitting.

iv) **Interpretability:**
The structure is easy to conceive: the **preliminary model** is aimed for **capturing the rough trend** of the "Danceability," and the **residual estimator** is designed for **learning the residual**, providing finer correction.

v) **Low Memory Consumption:**
The model does not consume very large memory: it is “trainable” on my desktop with 48 GB RAM, and only occupies relatively small memory space.

vi) **Overfitting Combating:**

- Boosting algorithm is generally less susceptible to overfitting.
- By limiting the number of boosting iteration and the depth of tree, overfitting can also be avoided.
- The limited fitting ability of linear regression may also act somewhat as regularization (reflected by the consistent MAE among training, validation and testing for preliminary model; see section 2.1.4, i)).

2.1.4 Training, Validation, and Testing

i) **Preliminary Model:**
The gradient-boosting-reinforced linear regression can only achieve roughly 1.82, 1.83, and 1.85 in MAE on training, validation and testing dataset. No significant improvement occurs after introducing the gradient boosting compared with normal linear regression. Nonetheless, this preliminary model does not overfit.

ii) **Pure GradientBoostingRegressor Model:**
To prove the enhancement comes from the residual learning instead of the versatility of GradientBoostingRegressor, we conduct a pure training of GradientBoostingRegressor, and the outcome suggests overfitting, whose Kaggle public MAE is 1.84, though its training and validation MAE can reach 1.14 and 1.66.

iii) **Hybrid with Error Estimator:**
By assembling the preliminary model and residual estimator together, magic happens: the hybrid models gives roughly 1.4 average MAE for training, 1.6 for validation, and 1.74 to 1.78 public MAE score on Kaggle, that is, a **0.1 MAE improvement**, compared to the preliminary model.

2.2 Model 2: Deep Learning

2.2.1 Methods

Besides traditional Machine Learning models, we have also tried models that are popular in Deep Learning world. Our models are mainly based on **Multilayer Perceptron (MLP)**; more specifically, with 4 hidden layers, where there are 126 neurons in each. We have chosen the **Rectified Linear Activation function (ReLU)** as our activation function and the **Least Absolute Error function (L1 Loss)** is used for our loss function. As for our optimization, several algorithms are implemented, such as **Stochastic Gradient Descent (SGD), Root Mean Square Propagation (RMSProp), and Adaptive Moment Estimation (Adam)**. Last but not least, two techniques are applied in order to **combat overfitting: Dropout and Early Stopping**. In particular, we implement Dropout with probability being 0.5 after every hidden layer and we execute Early Stopping if there is no improvement in our models after 5000 batches.

2.2.2 Model Characteristics

i) **Efficiency:**
While MLP is considered more “powerful” than many other traditional Machine Learning models, it is also considered more “complex”, for there are plenty of neurons and layers; due to its “complexity,” it needs more time than many other models for training undoubtedly.

ii) **Scalability:**
If we scale up the size of data, MLP will still yield praiseworthy results with its power, but we have to keep in mind that more data bring more time for training, especially when the data are “tremendous.”

iii) **Interpretability:**
Despite the fact that MLP is easy to design and visualize, it is pretty hard to analyze theoretically owing to its “black box nature,” not to mention interpreting the results.

To make a conclusion on MLP, we suggest that one be cautious when implementation, or he or she may end up spending plentiful time on training or not knowing what is happening during/after training.

2.2.3 Training, Validation, and Testing

Before we start our training, we have split the training set into two parts, 11.26% being the validation set and the remaining being the subtraining test. After the split, we standardize each feature in all data with the mean and the variance of the corresponding feature in the training set. We set our batch size to be 62 and the maximum number of epoch to be 100. Then we begin our training and validation process; note that we calculate the L1 Loss of the subtraining set and that of the validation set every 50 batches and the latter is used to determine whether to "early-stop" at that moment. We repeat our training and validation process so as to see which optimizer is the best. In our observation, SGD defeats the other two since it produces more stable results. Then we start to tune its parameters, which are **Learning Rate, Weight Decay, and Momentum**. Several values are used for tuning, such as 0.1126, 0.01126, 0.001126, and 0.0001126. After tuning, we have discovered that the best parameters for Learning Rate, Weight Decay, and Momentum are 0.001126, 0.1126 and 0.1126, respectively. In the end, we obtain an L1 Loss of 1.86284 in the testing set (public score on Kaggle), while we get 1.32454 for our best L1 Loss in the validation set.

2.3 Model 3: Ranking Regression Based on Binary Classification

2.3.1 Methods

- Overview
  Based on [LL06], we try to use the framework to convert this multiclass classification problem into a binary classfication problem with the extended examples. The reason for using this framework is that we believe "Danceability" has its natural order so that we can make use of its monotonicity. In other words, we view the results of "Danceability" as not just meaningless numbers but scores with respect to their corresponding songs. If we want to know how good a song $\mathbf{x}$ is, the associated problem will be "Is this song $\mathbf{x}$ better than k?", where $k \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ stands for "Danceability." If there is a change between $k = n$ and $k = n + 1$, we can confirm that the "Danceability" is n due to its natural monotonic order.

- Extended Examples
  Assume that we have an unknown multiclass classfication model f and an example $\mathbf{x}$ whose "Danceability" is n, then we can generate 10 extended examples with their newly assigned labels being 0 or 1. The original labels range from 0 to 9; if its original label is greater than a particular number in k, its newly assigned label will be True; otherwise, it will be False.

Algorithm 3 Extended Binary Classfication

```
Input : Original data set (x, y)
Output : A binary classification model $f_b$
1: for each $t \in [1, N]$ do
2:     for $k \in [0, 9]$ do
3:         $\tilde{x}_{n,k} \leftarrow (x_n, k)$
4:         $\tilde{y}_{n,k} \leftarrow [\![y_n \leq k]\!]$
5:     end for
6: end for
7: Train a binary classifier $f_b$ based on ($\tilde{x}$, $\tilde{y}$)
8: Return Binary Classifier $f_b$
```

2.3.2 Model Characteristics

i) **Efficiency:**
Since the extended dataset is $N \times K$, where N is the number of exteneded samples and K is the number of classes (10 choices here), it takes more time to train the binary classifier. Besides training stage, it also consumes more time in the prediction stage in the framework. It has to find the break point where the label changes from True to False. Even if we perform binary search to find the break point, it still needs $O(N * logK)$ in time complexity to do the prediction.

ii) **Scalability:**
The scalability in this framework is based on the binary classifier we choose. If the best binary classifier is sensitive to the number of features, then it will be significantly influenced if the number of features grows. Similarly, if the best classifier is sensitive to the number of examples, then it will also be influenced if the number of examples grows. For example, it is difficult for us to compute the inverse of a huge matrix.

iii) **Interpretability:**
This framework has very good interpretability because it is natural for us to deal with the multiclass classification problem. Suppose that you are going to give a rank to a movie from 1 to 10. Some may make decision via repeatedly asking his or her brain whether the movie is better than a particular score s or not. The binary classifier is to mimic our brain to get the final ranks.

2.3.3 Training, Validation, and Testing

We compare several binary classification models and record their performances in the following table.

We only use the training set for training and use MAE as E_{in}, while we use 0/1 error as E_{in} for extended examples. The result is shown in Table 2.

error\model	Decision Tree	KNN	Bagging classifier	AdaBoost classifier
Extended example	0.010781	0.08668026	0.006396	0.16737333
Danceability(Within training data)	0.1065812	0.61712288	0.07361677	1.6737332
Danceability(extra test data)	2.725832	3.486529	2.511885	2.220285

Table 2: Different Binary Classification Models Comparison without Testing Set

We submit the best three models and the MAE is 2.01425 in Bagging classifier, 2.31728 in Decision Tree, and 3.45929 in K-Nearest-Neighbors method, respectively. From the difference between the submission and testing, we conclude that there are several problems in this framework. First, it is not guaranteed that binary classifiers yield monotonic results on the extended examples generated from one specific example. Second, if the binary classifier overfits the data, the whole framework will be influenced because of the architecture in the framework. To summarize, converting the multiclass classification problem into a binary one is a simple thought and has an acceptable performance if we choose a good binary classifier by validtion, but it has its difficulty to yield better results than other models under this framework.

3 Conclusion

According to the performances of the three models above, we decide to recommend the first model, **Boosting Methods Combined with Residual Learning**, to you. The residual learning framework is popular in the field of Machine Learning nowadays [He16], so we try to combine it into our model and get brilliant results. However, you need to be aware of the time complexity in this model. Though the training time is roughly linear in D and N, the GradientBoostingRegressor itself is quite time-consuming with large T. Luckily, there is an alternative package HistGradientBoostingRegressor which can cut down the time consumption to one-tenth compared to the original method, though it will slightly sacrifice the accuracy (around 0.01 level difference for Kaggle pulic MAE). Due to its good interpretability, we can easily analyze the model performance in differenent stages and modify or improve our model until our expectation is reached. As for the overfitting problems, it is not serious for boosting algorithm, and well combated by limiting the depth of the regression tree. Hence, we do not have to worry about this problem at all.

4 Contribution

i) Data Preprocessing: Guan-Ming Su (Filling `NaN`), Tien-Leng Wu (Tokenization, One-Hot Encoding), Wei-Tang Chien(“Album” and “Track” processing)

ii) Model Training: **(a) Model 1:** Guan-Ming Su **(b) Model 2:** Tien-Leng Wu **(c) Model 3:** Wei-Tang Chien

iii) Report Writing: **(a) Drafting:** Wei-Tang Chien, Tien-Leng Wu, Guan-Ming Su **(b) Review and Correction:** Tien-Leng Wu

References

[He16] Zhang X. Ren S. Sun J. He, K. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 14, 2016.

[LL06] Ling Li and Hsuan-Tien Lin. Ordinal regression by extended binary classification. *Advances in neural information processing systems 19*, 14, 2006.