XGBoost Prediction of the Popularity of Songs

1. Introduction to XGBoost [1]

XGBoost stands for “Extreme Gradient Boosting”. It is an optimized distributed gradient boosting library, and is designed to be efficient and flexible. It implements machine learning algorithms under the gradient boosting framework, and provides parallel tree boosting.

When constructing a mathematical model for prediction from inputs xi to outputs yi, the parameters denoted by θ’s are what we should determine through training. For XGBoost, the performance of a model is expressed by an objective function, obj(θ). The objective function consists of two parts: the training loss *L* and regularization term Ω, . The training loss measures how predictive the model is with respect to the training data. A common choice of loss for regression is the mean squared error, . Another choice is the logistic loss, , which is commonly used in logistic regression. The regularization term is used to control the complexity of the model, which helps us to avoid over fitting.

XGBoost uses decision tree ensembles as the model choice. The decision tree ensemble model consists of a set of classification and regression trees (CART). While classical decision trees only contain threshold values, CART assigns each leaf a score (Fig.1). The values associated with each leaf are similar to weights and penalties, give interpretations beyond classifications and allow optimization.

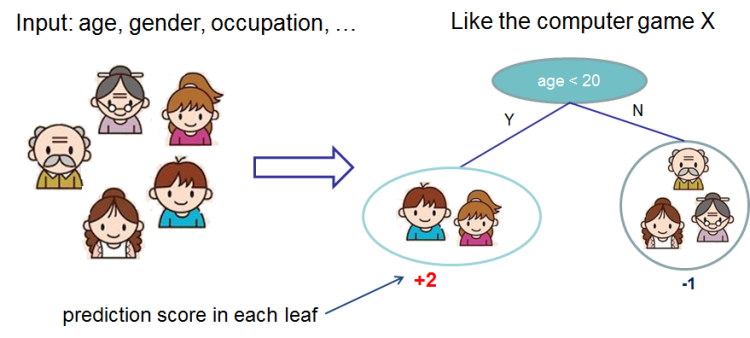
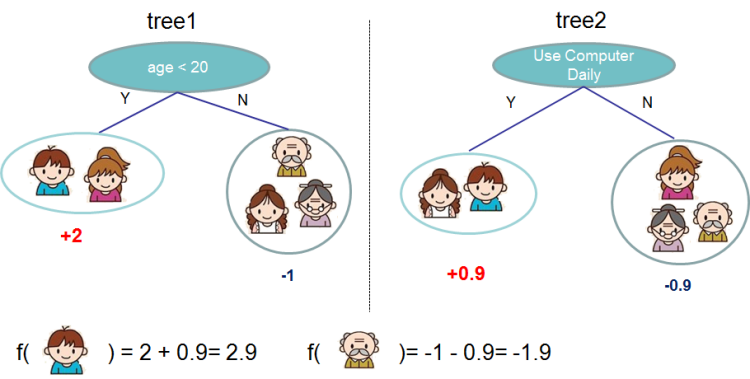


Fig.1 In CART, each leaf will have a score. For each object, the sum of all scores is the final score.

In application, predictions of multiple trees are combined to get an ensemble model. It is in principle similar to the random forest method. For each tree, an object will be assigned with a prediction score during classification. The sum of all prediction scores from all trees of an object is the final score. These leaf scores together with the tree structures are contained by a function  representing a tree. Then the objective function is expressed as, in which *l* is the loss, and ω is the complexity of the tree.

The tree training uses an additive strategy, add one tree each time to fix the result. Then we can optimize the objective function by changing how we calculate the loss and the complexity.

However, for some edge cases, this training method fails and results in a degenerate model, because only one feature dimension is considered at a time.

1. XGBoost Application
2. The First Model

Our data set consists of 102433 songs, each song has 18 features including the popularity we want to predict. After decomposing the data set into inputs and output, training and testing parts, we did an initial XGBoost modeling by setting all parameters to default (Fig.2). Here we took 20% of the entire data set as the test part.

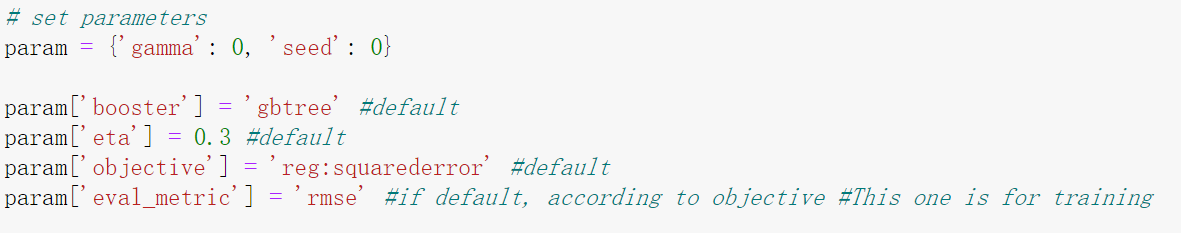


Fig.2 Parameters for the First Model

For XGBoost regression, we used the method “XGBRegressor” in the package “xgboost”. The predicted popularity based on test data was then compared with the true values. We used RMSE loss to evaluate the accuracy of the XGBoost model (Fig.3).

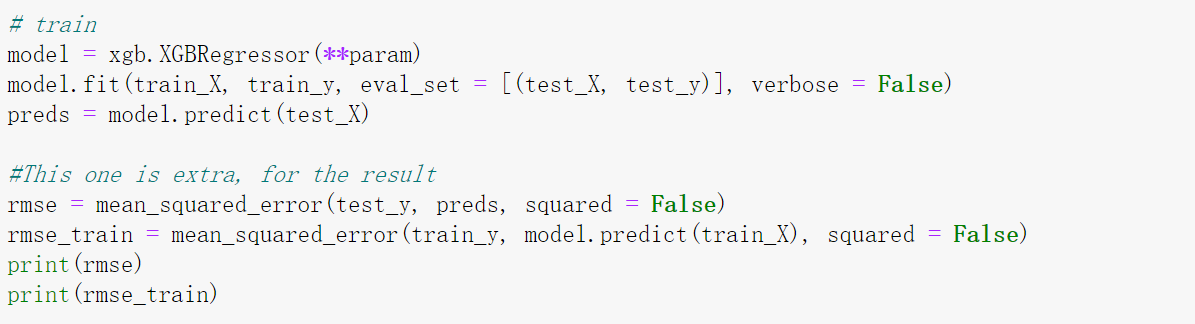


Fig.3 Training and Calculating RMSE of the First Model

The XGBoost model showed a better root mean square error value than linear regression model. The RMSE values for training and testing data are 10.152 and 10.266, respectively. Here testing RMSE would be lower if the model predicts more accurately, and training RMSE would be lower if the model overfits more.

From the plot (Fig.4) we can see that, the XGBoost model mostly followed the true values, and predicted very well. Due to a non-Gaussian distribution and lots of low-popularity songs, some data points at low popularity could not be covered by the model. But the difference distribution showed that most errors were no larger than 20.

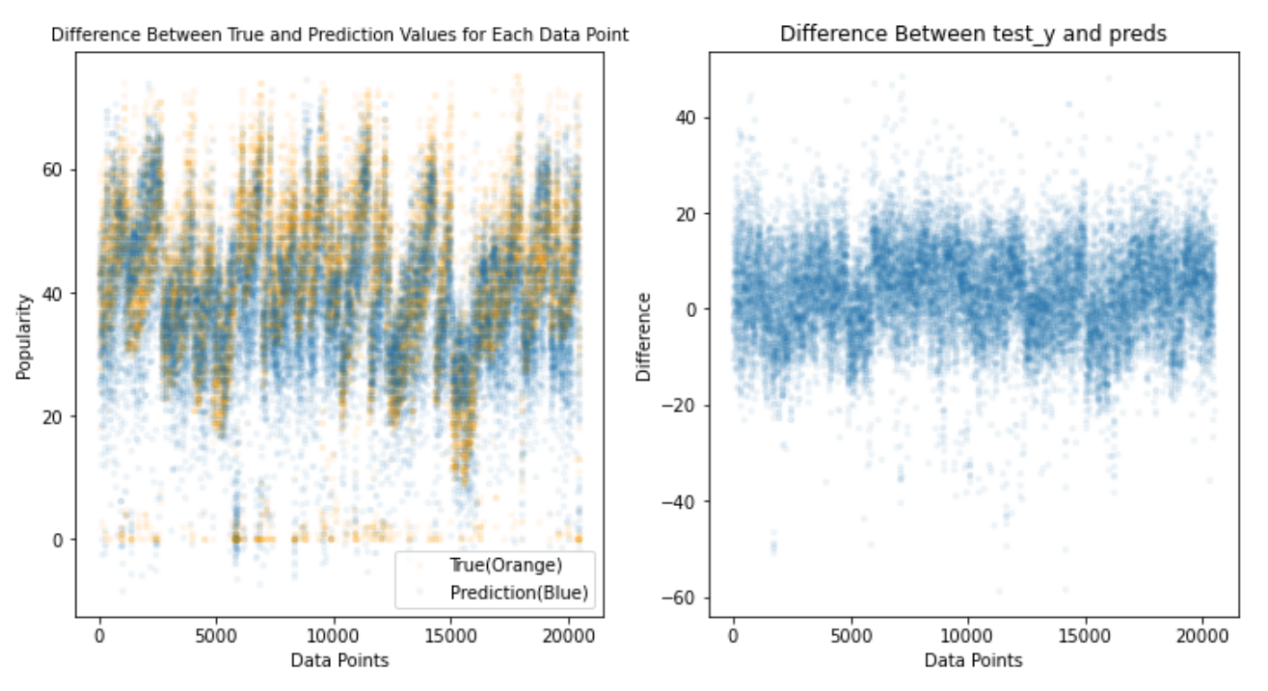


Fig.4 Difference Between True and Predicted Popularity

1. The Tuning of Parameters [2]

XGBoost has many parameters available for tuning. We chose several important ones, and plotted the change of RMSE value with respect of those parameters.

The first parameter is the maximum depth. It represents how deep a decision tree could develop. If the trees have too less layers, then the objects cannot be classified well. On the other hand, if there are too many layers, the model then will be more complex and may overfit. Plus, XGBoost aggressively consumes memory when training a deep tree. A proper training depth will influence the model and thus the results.

The default maximum depth for XGBoost is 6. After applying several maximum depths to the model, with other parameters set to default, we obtained the RMSE plot (Fig.5). It can be seen that, the training RMSE decreases when the depth increases from 1 to 6. while the testing RMSE reaches a minimum at depth equals 5. From our view, we think that after two lines intersect, the prediction loss for training data itself is smaller than prediction loss for new data, that means somehow an overfitting model. Thus, we chose 5 as the optimized maximum depth. If not tuned, it would be 6.

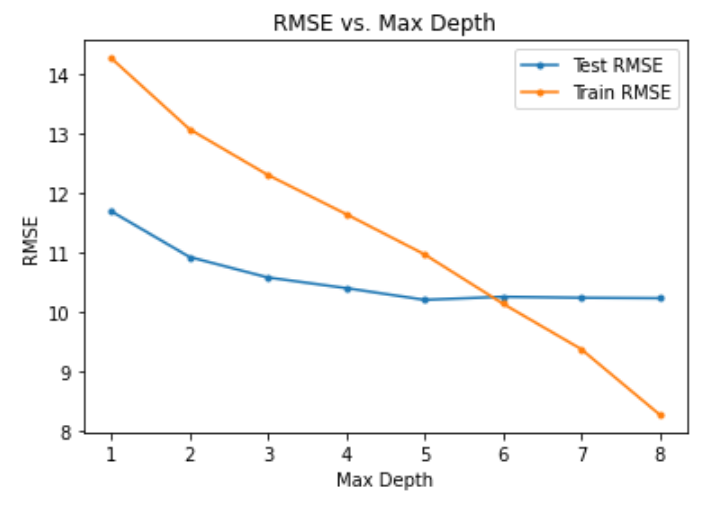
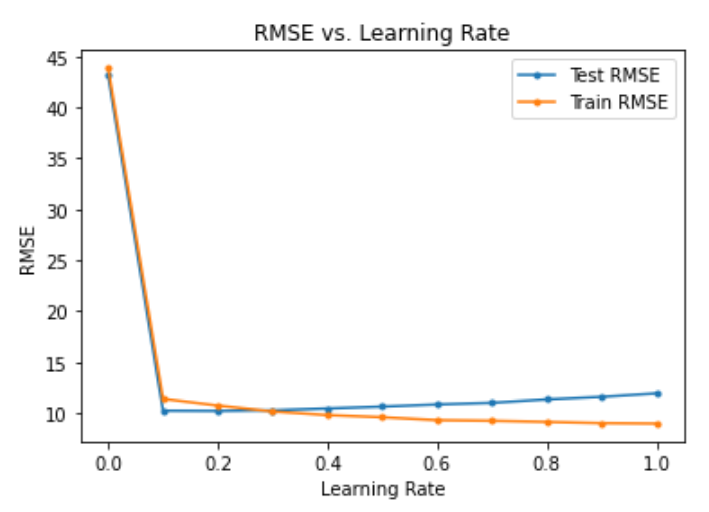
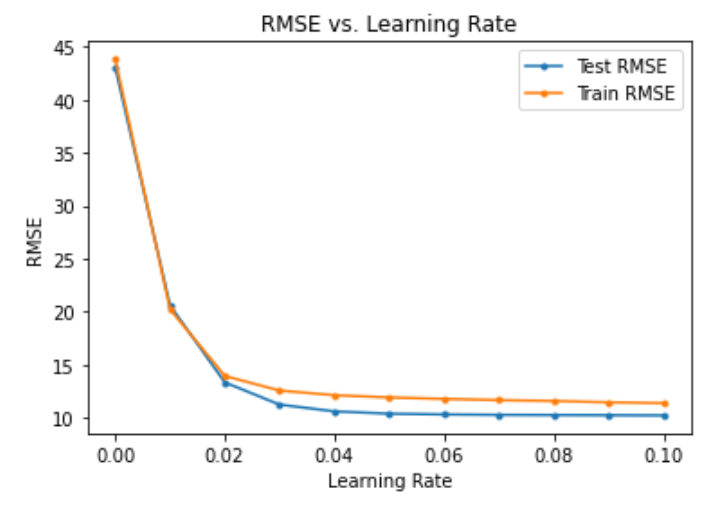


Fig.5 Maximum Depth Tuning

The second parameter is the learning rate. It is the step size shrinkage used in updates to prevent overfitting. After each boosting step, we can directly get the weights of new features, and the learning rate shrinks the feature weights to make the boosting process more conservative.

We first tested the effect of learning rate in the range of 0-1 with step size of 0.1. The curve of RMSE shows a significant turning point between 0 and 0.1. Next, we tested again in the range of 0-0.1 with the interval of 0.01. The curve of RMSE still has a turning point, but now it can be determined by using the elbow method. The ideal value of the learning rate should be about 0.02, because after this point, the decline rate of RMSE slows down significantly. The two curves are showed as below (Fig.6).

Fig.6 Learning Rate Tuning



The third parameter is the minimum split loss. It is the minimum loss reduction required to make a further partition on a leaf node of the tree. The larger the value is, the more conservative the algorithm will be.

We first tried to tune minimum split loss in the range of 0-10, but the result did not show anything obvious, so we extended the range to 0-50 with the step size was 0.01. The plot shows that the change of RMSE is very unstable with the increase of loss rate (Fig.7). In the range of 0-50, we estimated that the most appropriate loss rate was about 36 because the testing RMSE was the lowest.

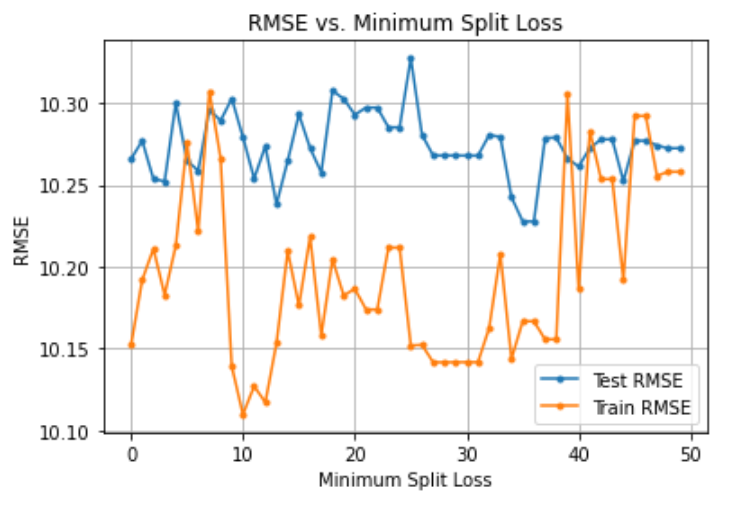


Fig.7 Minimum Split Loss Tuning

The fourth one, the tree method, is automatically set by XGBoost. However, we tried different available tree methods and plotted their RMSE loss. The result was simple: XGBoost chose “exact” as the combination of commonly used updater (Fig.8).



Fig.8 Tree Method Tuning

The fifth one is the subsample ratio of the training instances. Setting this value to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. The non-one value of subsample ratio can prevent overfitting. Subsampling will occur once in every boosting iteration.

The result is showed in Fig.9. The RMSE loss showed a very clear curve. After we trained with step size equaled to 0.01, we decided to use 0.001 as the step size and trained again from 0 to 0.01. The elbow method gave that, 0.01 is the optimized ratio of subsampling.

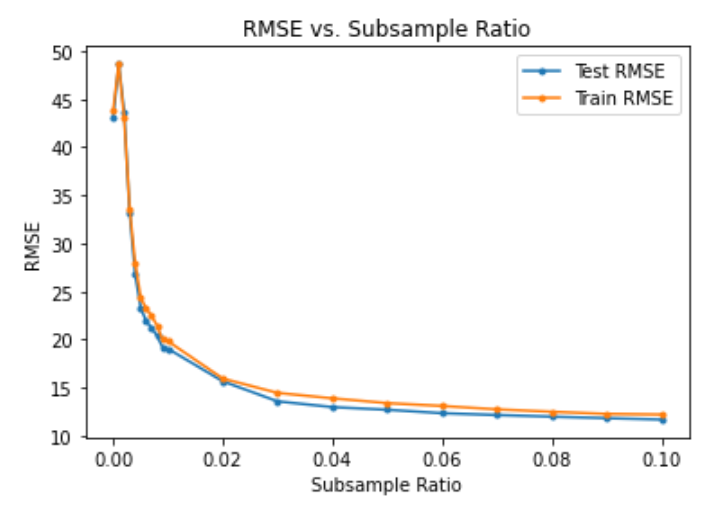


Fig.9 Subsample Ratio Tuning

The sixth one is the dropout rate. It determines the fraction of previous trees to drop during the dropout. Since the RMSE curve had variations between 0 and 0.1, we again used a smaller interval of 0.01 in 0-0.1, and got a minimum RMSE loss at 0.03 (Fig.10). This part of tuning needed a lot of calculations, and it consumed memory very aggressively during the training.

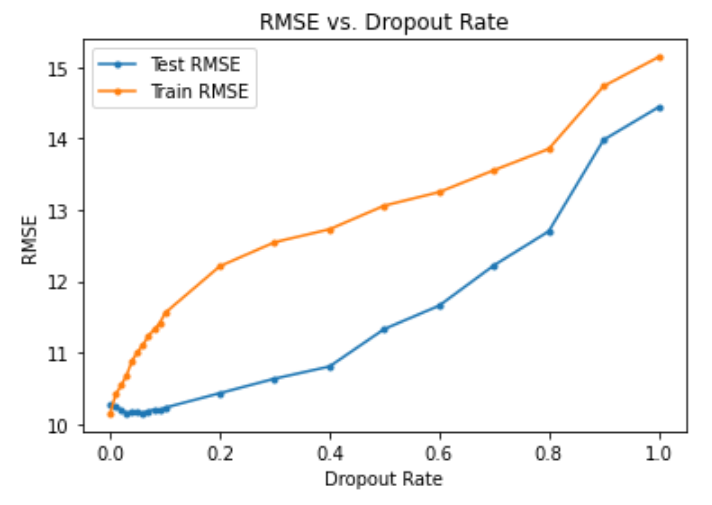


Fig.10 Dropout Rate Tuning

After tuning these parameters, we tried to combine them together and calculate the RMSE loss again. However, if we apply all the optimized parameters, the result actually will get worse, change from about 10 to about 16-17. Moreover，we tried to not fix the learning rate and subsample ratio (Fig.11), and had a better result (Fig.12).

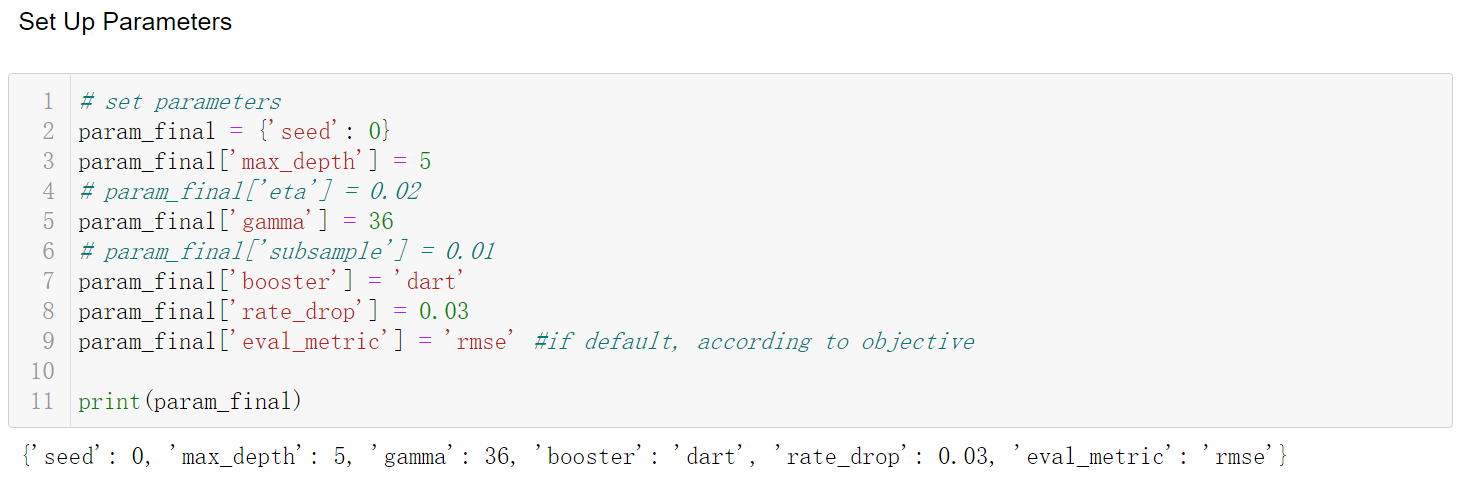


Fig.11 Final Model Tuning

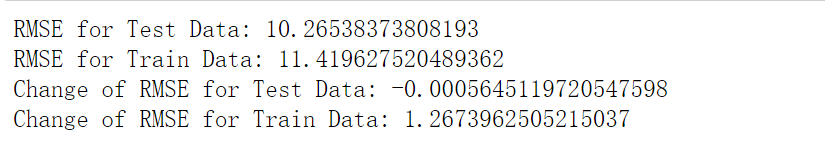


Fig.12 Final Results

1. Final Thoughts

The XGBoost model was a very good model from the beginning. Our tuning just raised the “training” loss a small fraction. We thought that means the model became less overfitting. Our final conclusion about using XGBoost method is, this method performs well with various automatically set parameters. We can tune and fix some of those parameters, but most of the time XGBoost itself would choose the best parameter for modeling.

References:

1. <https://xgboost.readthedocs.io/en/stable/tutorials/model.html>
2. <https://xgboost.readthedocs.io/en/stable/python/python_intro.html#setting-parameters>