ML Final Project: Predicting Danceability

The Economists

June 2023

1 Data Preprocessing

For the purpose of testing different data preprocessing methods, we have selected L1 regularized linear regression (LASSO) as our testing model.

Step 1: Transform Data

Logarithmic Transform: By plotting the distributions of the data, we can see that the feature *Views*, *Likes*, *Stream*, *Comments* and *Duration* are suitable for $log(1 + x)$ transform.
Categorical Data: Meanwhile, *Album Type*, *Licensed*, *Official video* and *Key* are categorical data, so we use one hot encoding on them.
Other Non-numerical Features: We drop other non-numerical features for its complex nature.

Step 2: Impute Missing Data

We use k-nearest neighbor(KNN) imputer with $N = 5$ to impute for missing value. Compared with simply filling the missing values with mean, the KNN imputer achieve a better MAE for approximately 0.1. The cross validation result with different L1 coefficient is summarized in the table below.

Step 3: Implement Polynomial Feature

Starting with no transform, we conduct cross validation with different L1 coefficient and increase the order respectively. The result is summarized in the table below. It is easily to see that the E_{in} is significantly lower using polynomial feature with degree = 2. However, the error increased drastically with degree = 3, which indicates overfitting. Thus, we use polynomial feature with degree = 2 as our polynomial transformation for LASSO regression and other models.

Imputation	Poly	$\lambda = 10^{-6}$	$\lambda = 10^{-3}$	$\lambda = 1$	$\lambda = 10^{3}$	$\lambda = 10^{6}$
mean	2	1.86812	1.86810	2.46631	2.51313	2.51313
KNN	1	1.88972	1.88972	2.42289	2.51313	2.51313
KNN	2	1.74705	1.74703	2.41371	2.51313	2.51313
KNN	3	3.31324	2.05598	2.41371	2.51313	2.51313

Step 4: Normalize Data

To overcome the convergence issue, we normalize each feature such that the mean of each feature is 0 and standard deviation is 1.

2 Models

Model 1: LASSO Regression

While pure linear regression is not strong enough, we choose LASSO (Least Absolute Shrinkage and Selection Operator) regression as one of our model. LASSO regression performs both variable selection and regularization to improve the model's predictive performance. We make feature transformation before the data be inputted into the model. We test the weight of punishment of [0.001, 0.01, 0.1, 1, 10, 100, 1000] to obtain the best model through cross-validation.

Efficiency

The running time of the LASSO regression algorithm depends on the number of features and the size of the dataset. Since LASSO performs feature selection by shrinking some coefficients to zero, it can effectively handle high-dimensional datasets with a large number of features. The computational complexity of LASSO regression is higher since it needs to test numerous coefficients' combinations.

Scalability

LASSO linear regression can be parallelized to enhance scalability. The computations involved in solving the LASSO optimization problem can be distributed across multiple cores or machines. By leveraging parallel processing, we can significantly reduce the training time for large datasets.

Popularity

LASSO regression has gained significant popularity in the field of machine learning, especially in scenarios where feature selection is crucial. Its ability to automatically shrink irrelevant features to

zero makes it suitable for tasks with a large number of predictors.

Interpretability

LASSO linear regression provides interpretable results by assigning zero coefficients to irrelevant features. This property enables us to identify the most important predictors contributing to the target variable. The regularization term in LASSO encourages sparsity in the coefficient estimates, which leads to a more interpretable model.

Model 2: Random Forest Regressor

Because the result is more similar to a continuous value, we used a random forest regressor as one of the models. We tried this model without polynomial transformation and with a second-order polynomial transformation.

Efficiency

The training speed depends on two factors: N, which represents the number of trees, and D, which represents the maximum depth of the trees. The more trees and higher depths we use, the longer it takes to complete the training process. In our testing, we experimented with values of N ranging from 1 to 200 (specifically [1, 2, 5, 10, 15, 20, 50, 100, 200]). Before implementing parallelization, the training process took approximately 8 hours to complete.

Scalability

Random Forest supports parallel computing due to its nature of being based on the bagging method. Unlike boosting, which trains based on the last training result, bagging allows for the parallel training of multiple trees simultaneously. This parallel computing capability enables Random Forest to take advantage of efficient computation and accelerate the training process.

Parameters

We used cross-validation to evaluate our parameters. Initially, we focused on determining the optimal maximum depth for the Decision Tree Regressor to avoid potential overfitting concerns. We find that when $D \geq 7$, the error increases., so we decide to use $D = 7$ as our final parameter. For optimal number of trees, we observed that the performance did not significantly improve when $N \geq 15$. Consequently, we concluded that using $N = 15$ yielded the best outcome and we selected these values as our final parameter settings.

Training Result

The E_{in}, which is the MAE of the training data, was 1.8. The submission result(public score) was about 2.223, private score = 2.505.

Model 3: XGBoost

Introduction

XGBoost[1] is a model based on the Gradient Boosted Decision Tree (GBDT) model, but with a different objective function. While GBDT simply uses the first-order gradient, XGBoost uses a second-order Taylor expansion and includes a regularization term in the objective function to prevent overfitting. The objective function of XGBoost in the original paper is as follows:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t)$$

where l is a convex differentiable loss function , y_i and $\hat{y}_i$ are the target value and the predicted value. The regularization term is:

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda ||w||^2$$

T is the number of leaves of the tree, w is the weight of leaves. Additionally, XGBoost differs from GBDT in terms of the score function used to evaluate and identify the optimal split in each subtree. Instead of using Gini impurity, XGBoost use a different score function for this purpose.

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T} \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} + \gamma T$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)})$, $h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)})$, I_j is the instance set of leaf j and q is a tree that maps each example to a specific leaf. However, the objective function of this project is MAE, which is not differentiable, so we tried two ways to solve this:

(1). Run regression with Pseudo Huber Loss, which is a twice differentiable alternative to absolute loss.

(2). Use the absolute error that XGBoost provide, but it won't guarantee optimal in distributed training. we use the option 2. because its performance is better and we tested to train it with number of thread = 1 to avoid bias, but the result do not differ from the result after parallelization. The testing result is shown in the Parameter session.

Efficiency

The most time-consuming aspect of XGBoost is sort each entries of each feature in order. Since real-world data is often sparse, XGBoost adopts a format known as CSC (compressed column)

to store the values. By utilizing CSC and storing the data in a sorted manner, XGBoost can achieve reusability and reduce time complexity. This efficiency improvement can be verified through amortized analysis. Furthermore, XGBoost incorporates a cache-aware prefetching algorithm to further enhance its speed. With the default setting of the maximum number of threads on our computer, it took approximately 5 minutes to train each parameter. Impressively, it only required 7 seconds to train the entire dataset and generate the submission result.

Scalability

Though traditional boosting methods generally do not support parallel computing. In XGBoost, the calculation of the score function for each CSC can be parallelized, enabling XGBoost to take advantage of parallel computing and accelerate the training speed. By default, XGBoost utilizes the maximum number of threads available on the system to parallelize the process and improve efficiency during training.

Parameters

We used cross-validation to evaluate our parameters. The default set of XGBoost uses L2-regularization, so we set $\lambda = 0$ to cancel it. Specifically, we tested different tree depths (D) ranging from [1, 2, 3, 4, 5, 7, 10, 15, 20] and coefficients of the L1-regularization term (α) from [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000].
We tried using Pseudo Huber error in the objective function, we found that the optimal values were $D = 4$ and $\alpha = 1000$. The corresponding E_{in} was 1.95, and the submission result produced a public score of 2.561.
We also tried the absolute error in the objective function, the optimal values were $D = 5$ and $\alpha = 1$. This yielded an E_{in} of 1.43, and the submission result achieved a public score of 2.2895.
We observed that the lowest error occurred when we use absolute error as the objective function with D set to 5 and α set to 1000. Consequently, we selected these values as our final parameter settings.

Training Result

The E_{in}, which is the MAE of the training data, was 1.43. The submission result(public score) was 2.2895, the private score is 2.585.

3 Comparison

Model	E_{in}	Public	Private	Training time	Other issues
LASSO	1.8441	2.1279	2.4853	15.8 seconds	Limited extensibility
Random Forest Regressor	1.6933	2.1609	2.4462	4 min and 10 sec	The training time is too long
XGBoost	1.4332	2.273	2.5928	6 seconds	Underlying overfitting concern

Both of these models have decent Scalability. These models are capable of handling larger datasets or larger amount of features efficiently. It can be seen that while XGBooost has the lowest E_{in}, its performance on public and private testing sets are not good. We believe this arise from the complex nature of the model, which may induce overfitting.

4 Conclusion

Based on our evaluation, we recommend using LASSO regression due to its accuracy and reasonable complexity and speed. While Random Forest Regressor and XGBoost may achieve low E_{in}, we acknowledge that these models can be complex, and our ways of finding the optimal tuning parameters might be too naive, resulting in poor submission scores. Furthermore, we acknowledge that we didn't effectively overcome the overfitting issue with only cross-validation and regularization. One possible approach to avoid overfitting is to adjust the coefficient on the regularization term and explore its impact on improving prediction performance. In addition to parameters, we should also consider testing different transformations for each model.

5 Workloads

Szu-Yu Lu: Data preprocessing, LASSO regression.
Chuan-Chi Hsu: LASSO Regression.
Yan-Xiang Chiu: Random Forest Regressor, XGBoost.

6 References

[1] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.