CS150A Database
Course Project

Yijie Fan
ID: 2020533120
fanyj@shanghaitech.edu.cn

Hailiang Zhu
ID: 2020533061
zhuhl2@shanghaitech.edu.cn

Abstract

In our project, we followed the guideline from instructor and went through data exploration and data cleaning. We then preprocessed the csv files with several methods in feature engineering. With those data, we trained our prediction model with seven common machine learning algorithms and select three of them to do extra hyperparameter tuning. We eventually got a great RMSE for 0.34725 with the LightGBM algorithm. PySpark implementation was also included in our project.

1 Explore the dataset

After data exploration we found that all columns can be divided into two types: categorical features and numerical features. The column of "Problem Hierarchy" is a combination of "Problem Section" and "Problem Unit". So we can split this column into two new columns in Feature Engineering part.

Since the number of the columns is too large, so we only considered several important columns that is strongly related to "Correct First Attempt". These columns contains: "Anon Student Id", "Problem Name", "Problem Hierarchy", "Problem View", "Step Name","KC(Default)", "Opportunity(Default)", "Correct First Attempt", "Correct Step Duration (sec)" and "Error Step Duration (sec)". We further explore the distributions of these columns, which are shown below:

Column	count	unique	freq
Problem Name	232744	1021	6166
Problem Hierarchy	232744	138	10225
Step Name	232744	60709	7354
KC(Default)	173489	348	18682
Opportunity(Default)	173489	36784	2619

Table 1: Distribution of Categorical Features

Column	count	mean	std	min	max
Problem View	232744	1.602838	1.5155546	1	21
Correct First Attempt	232744	0.780935	0.413613	0	1
Correct Step Duration (sec)	181599	17.924024	35.179534	0	1067
Error Step Duration (sec)	50853	60.547204	89.287960	0	1888

Table 2: Distribution of Numerical Features

2 Data cleaning

To clean the data, we mainly remove outliers and check missing values.

Removing outliers: In order to detect whether a value is an outlier, we set a test formula, that is, whether the difference between the current value and the average value is more than 20 times the standard deviation, and if it exceeds it, it is considered as an outlier and we would drop it.

Checking missing values: It can be observed that some rows of columns have missing data, some are reasonable, for example, some students finish the question in the first step, then "Hint" will be NaN, some are not so reasonable, it will affect the extraction in Feature Engineering and the final prediction, these NaN will be removed.

3 Feature engineering

Feature engineering is one of the most important parts of this project. To solve this problem, our work mainly contains five parts, which are Removing, Splitting, Encoding, Compression, and Building new columns.

Removing: We mainly removed several columns that are not provided in the test set, these columns contain Incorrects, Hints, and so on. We also removed several columns which are considered useless during the Data Exploration part, these columns contain Error Step Duration and so on.

Splitting: We split several "combined" columns such as "Problem Hierarchy" into two single columns which are "Problem Unit" and "Problem Section".

Encoding: We mainly encode the categorical features. However, the number of features is too large to use one hot encoding method discussed in class. Therefore, we used a very naive method which is to encode the features by the index(the index start from 1 and increase 1 each time we see a new value).

Compression: In the Compression part, instead of encoding the features, we compressed the columns. This is because encoding cannot best reflect the attribute of the columns. For example, we compressed the "KC(Default)" by counting the number of KCs to measure the difficulty of the problem. We also did this to "Opportunity(Default)" by finding the minimum and average of this column to measure the probability of whether the student can correctly answer the question. We also noted that several features in the "Step Name" column are only basic arithmetic operations, so we compressed these features into the "basic op" feature.

Building new columns: We generate more high-level columns for predicting: CFAR which is the correct first attempt ratio. We calculated the CFAR of "Step Name", "KC(Default)" and so on. We newly build this column because we can more easily observe the relationship between the problem difficulty and CFA and the relationship between the students' discrepancy and CFA.

The columns listed below are the final columns we used in Learning part after feature engineering:

Problem View	Correct First Attemp	KC num	Opportunity Average
Opportunity Min	Anon Student Id	Problem Name	Problem Unit
Step Name	Anon Student Id CFAR	Problem Name CFAR	Problem Unit CFAR
Problem Section CFAR	Step Name CFAR	KC(Default) CFAR	Problem Section

Table 3: column used eventually

4 Learning algorithm

For such a machine learning problem with so many features, it's hard to identify it as a regression problem or a classification problem. So when involving algorithms having both regression and classification method, we would try both of them to see which one has a better performance. Here we will use the default hyperparameter setting for all algorithms and tune the hyperparameters of those performing well.

Here, the algorithms we chose are Adaboost, Decision Tree, Random Forest, XGBoost, LightGBM, K-Nearest-Neighbours(KNN) and Multilayer Perception(MLP). They are all popular machine learning algorithms and we believe we can yield acceptable classification results from them. We will give a brief description of each algorithm.

Adaboost: Short for Adaptive Boosting, is a statistical classification meta-algorithm. We used both `AdaBoostRegressor()` and `AdaBoostClassifier()` to test which one is better. The former one wins.

Decision Tree: A decision support tool that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements. We used `tree.DecisionTreeClassifier()` as the model.

Random Forest: An ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time. We used both `RandomForestRegressor()` and `RandomForestClassifier()`. The former one wins.

XGBoost: eXtreme Gradient Boosting, an open-source software library which provides a regularizing gradient boosting framework for various programming languages. We used `XGBClassifier()` as the model.

LightGBM: Short for light gradient-boosting machine, is a free and open-source distributed gradient-boosting framework for machine learning. We used both `lightgbm.LGBMRegressor()` and `lightgbm.LGBMClassifier()` to test which one is better. The former one wins.

KNN: Is a non-parametric supervised learning method, used for regression and classification problems. We used both `neighbors.KNeighborsRegressor()` and `neighbors.KNeighborsClassifier()`. The former one wins.

MLP: is a fully connected class of feedforward artificial neural network (ANN). We used `MLPRegressor(hidden_layer_sizes=(100, 5, 100), activation='tanh', solver='adam')` as the model.

The result is shown as below. From the table below, we can easily observe that **Random Forest** outperforms all the other algorithms. And we decided to choose Adaboost, Random Forest and LightGBM for hyperparameter selection in section 5.

Method	RMSE
AdaBoost	0.38379
Decision Tree	0.45849
Random Forest	0.35080
XGBoost	0.39517
LightGBM	0.35374
KNN	0.40157
MLP	0.37152

Table 4: RMSE

5 Hyperparameter selection and model performance

Though we got relatively great RMSEs using default hyperparameter settings in several algorithms, we decided to use GridSearch to tune the hyperparameters for **Adaboost, Random Forest and LightGBM** to see if there still exists better RMSE.

A brief introduction to GridSearch: GridSearchCV is the process of performing hyperparameter tuning in order to determine the optimal values for a given model. By passing the parameters you want to tune as parameter grid into the function `GridSearchCV()`, it will automatically check all combinations of the parameters you input and output the one that fits best.

AdaBoost: We implement GridSearch on `AdaboostRegressor()`'s 2 parameters: 'n_estimators' and 'learning_rate', and yield the result of of learning_rate=0.11473684 and n_estimators=28 after 200 minutes of training. The corresponding RMSE for this hyperparameter setting is 0.37962, better than the default setting. But we also tried to manually set the 2 parameters as learning_rate=0.01 and n_estimators=50, its RMSE is 0.37336. We think this may due to the fact that the scale we chose for GridSearch for AdaBoost wasn't that proper, so we only found a local optimal answer.

Random Forest: We implement GridSearch on `RandomForestRegressor()`'s 1 parameter: 'n_estimators' and yield the result of n_estimators=150. The corresponding RMSE for this hyperparameter setting is 0.35304. We also manually set the n_estimators=100 and yield a better RMSE for 0.35351. We think this may also due to local optimization.

LightGBM: We implement GridSearch on `lightgbm.LGBMRegressor()`'s 2 parameters: 'num_leaves' and 'n_estimators' and yield the result of num_leaves=150 and n_estimators=100. The corresponding RMSE for this hyperparameter setting is 0.34725, which is quite desirable.

So we get the minimum RMSE as 0.34892 from the LightGBM algorithm. The below table is showing the tuning result.

Method	original RMSE	new RMSE with tuned hyperparameters
AdaBoost	0.38379	0.37336
Random Forest	0.35080	0.35351
LightGBM	0.35374	0.34725

Table 5: hyperparameter tuning

6 PySpark implementation (optional)

Pyspark is particularly efficient when we want to deal with large amounts of data. We mainly use pyspark in the Feature Engineering step, because this step involves a lot of repetitive and simple operations on a large amount of data, and using pyspark can save a lot of time and improve efficiency. Specifically, we first used SQLcontext to generate the dataframe, then we designed algorithms for the columns that need to be modified in the Feature Engineering step, then we used the pyspark interface to convert these algorithms into user defined functions, and finally we used these user defined functions for the columns that need to be modified.