CS150A Database
Course Project

Student 1: Entropy-Fighter
ID: 2020533xxx
xxxx@shanghaitech.edu.cn

Student 2: Chenyh
ID: 2020533xxx
xxxx@shanghaitech.edu.cn

Guideline

This project is about using ML to predict student performance. In the project, we follow the 5 steps given by the instructor and use PySpark to improve the efficiency. Our final RMSE on valid data in "test.csv" is close to 0.3539.

1 Explore the dataset

1.1 Getting basic information by reading the dataset

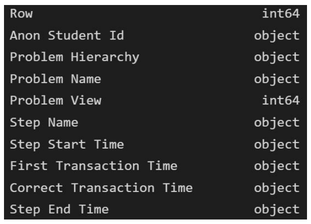
In our data set, we can find that the total 19 features can be separated into 2 types: categorical features like "Student Name", and numerical features like "Step Duration".

As for the structure of data, we find "Problem Hierarchy" can be separated into "Problem Unit" and "Problem Section". We also find one row can have more than one KC, which are connected by $\sim\sim$. Therefore, we can separate the KCs by $\sim\sim$. "Opportunity"'s situation is the same with "KC".

As for the missing values, in "train.csv", we find some values of "Correct Step Duration", "Error Step Duration", "KC" and "Opportunity" are NaN. In "test.csv", we find that values of "Step Start Time", "First Transaction Time", "Correct Transaction Time", "Step End Time", "Step Duration", "Correct Step Duration", "Error Step Duration", "Correct First Attempt", "Incorrects", "Hints" and "Corrects" are missing.

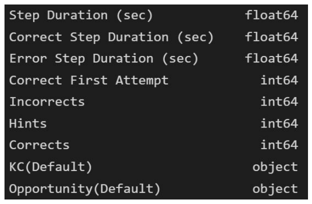
1.2 Data type of each feature

Using dtypes, we get the following results.



```
Row                            int64
Anon Student Id               object
Problem Hierarchy             object
Problem Name                  object
Problem View                   int64
Step Name                     object
Step Start Time               object
First Transaction Time        object
Correct Transaction Time      object
Step End Time                 object
```

(a) type

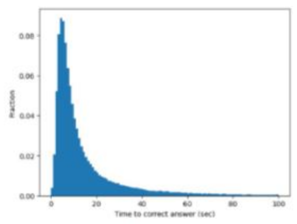


```
Step Duration (sec)            float64
Correct Step Duration (sec)    float64
Error Step Duration (sec)      float64
Correct First Attempt            int64
Incorrects                       int64
Hints                            int64
Corrects                         int64
KC(Default)                     object
Opportunity(Default)            object
```

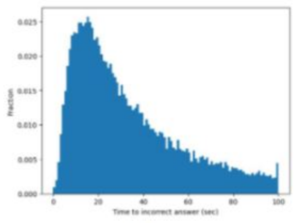
(b) type

1.3 Visulization of some statistics

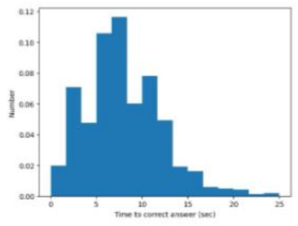
From the figure below, we can find that if the step duration is long, the CFA tend to be incorrect.



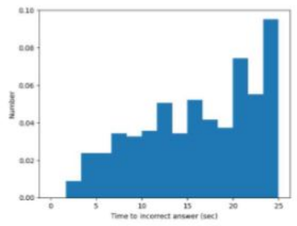
(c)



(d)



(e)



(f)

2 Data cleaning

In this section, firstly, we remove some meaningless NaN or replace some NaN to 0 for easier caluculation. Then, we make sure every row is unique and drop duplicates. The outlier is another thing that we need to consider. We do a easy detection to check whether there are some singular values which are inconsistant with the normal values.

After doing these steps, we simplify the data set for following sections according to our needs. As is introduced in section 1, many column values are missing in "test.csv". Therefore, we consider them useless for this project(maybe few features are meaningful, but we don't consider them for the convenience) We remove the following columns in "train.csv": ["Row", "Step Start Time", "First Transaction Time", "Correct Transaction Time", "Step End Time", "Step Duration (sec)", "Correct Step Duration (sec)", "Error Step Duration (sec)", "Incorrects", "Hints", "Corrects"]

3 Feature engineering

This part is very important for lowering the RMSE. In the data cleaning part, we have removed some irrelevant features. In this part, we modify some categorical features and add some CFARs as new features.

3.1 Feature modifications

Firstly, we separate "Problem Hierarchy" into "Problem Unit" and "Problem Section" according to our findings in data exploration. Secondly, we should compress the information of "KC" and "Opportunity". The data format of "KC" is like "String$\sim\sim$String$\sim\sim$String", and the data format of "Oppportunity" is like "Integer$\sim\sim$Integer$\sim\sim$Integer", which are too complex. We write 2 functions to simplify them: getKCCount(kcs) and getOpportunityAverage(oppo). The function names give good explanation of what we do. We use these 2 functions' results to replace the orginal 2 columns' values.

3.2 Converting categorical features to numerical ones

As is shown in data exploration section, we have both categorical features and numerical features. In order to better fit the model, we need to convert categorical features to numerical ones. Therefore, we need to do some operations on the following columns: ["Anon Student Id", "Problem Name", "Problem Unit", "Problem Section", "Step Name"]. We should do the operations on the union of train set and test set, otherwise we would have 2 different conversions. We use python's dictionary for our naiive conversion. We assign values starting from 0 to different objects.

3.3 Adding new features

To better predict the CFA, we calculate different kinds of Correct First Attempt Ratio(CFAR) and add them as new features(We get this idea from the paper: [2010]Feature Engineering and Classifier Ensemble for KDD Cup 2010, Hsiang-Fu Yu). The formula of CFAR is shown below.

$$featureX\ CFAR = \frac{number\ of\ rows\ with\ featureX\ =\ xid\ and\ CFA\ =\ 1}{number\ of\ rows\ with\ featureX\ =\ xid}$$

By the above formula, we write the function getCFARTemplate() and calculate "Student CFAR", "Problem CFAR", "Unit CFAR", "Section CFAR", "Step CFAR" and "KC CFAR", since we assume these "featureX" like "student" and "problem" are relevant to the CFA.

4 Learning algorithm

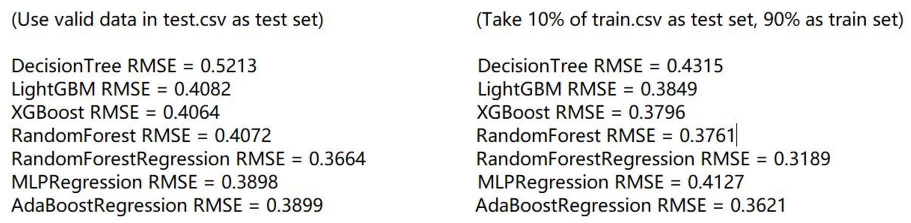
After reading some papers and other people's previous work, we decide to choose one best learning algorithm among the following methods.

- DecisionTreeClassify: A Decision tree is a tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label. A tree can be "learned" by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called recursive partitioning.
- LGBMClassify: Light GBM is a gradient boosting framework that uses tree based learning algorithm. While other algorithms trees grow horizontally, LightGBM algorithm grows vertically meaning it grows leaf-wise and other algorithms grow level-wise. LightGBM chooses the leaf with large loss to grow. It can lower down more loss than a level wise algorithm when growing the same leaf.
- XGBoostClassify: In this alogorithm, decision trees are created in sequential form. Weights play an important role in XGBoost. Weights are assigned to all the independent variables which are then fed into the decision tree which predicts results. The weight of variables predicted wrong by the tree is increased and these variables are then fed to the second decision tree. These individual classifiers then ensemble to give a strong and more precise model.
- RandomForestClassify: Random forest computes on multiple decision trees and involves the bagging theory. Random Forest has multiple decision trees as base learning models. We randomly perform row sampling and feature sampling from the dataset forming sample datasets for every model.
- RandomForestRegression: The principles are the same with RandomForestClassify, but we use regressor rather than classifier.
- MLPRegression: Neural networks can sometimes solve complex problems in machine learning. And in this semester, we learn the multi layer perceptron in course CS181, so we want to use mlpregressor in sklearn to have a try.
- AdaBoostRegression: What Adaptive Boosting does is that it builds a model and gives equal weights to all the data points. It then assigns higher weights to points that are wrongly classified. Now all the points which have higher weights are given more importance in the next model. It will keep training models until and unless a lowe error is received.

We feed training data into these models, then predict the CFA of testing data. Finally, we compare our prediction with the correct answers and calculate RMSE. The RMSE results of the above methods are shown below.

It is obvious that RandomForestRegression has best performance among all these methods. We explain our possible reasons below.

Every decision tree has high variance, but when we combine all of them together in parallel then the resultant variance is low as each decision tree gets perfectly trained on that particular sample data and hence the output doesn't depend on one decision tree but multiple decision trees. The basic idea behind RandomForestRegression is to combine multiple decision trees in determining the final output



```
(Use valid data in test.csv as test set)        (Take 10% of train.csv as test set, 90% as train set)

DecisionTree RMSE = 0.5213                      DecisionTree RMSE = 0.4315
LightGBM RMSE = 0.4082                          LightGBM RMSE = 0.3849
XGBoost RMSE = 0.4064                           XGBoost RMSE = 0.3796
RandomForest RMSE = 0.4072                      RandomForest RMSE = 0.3761
RandomForestRegression RMSE = 0.3664            RandomForestRegression RMSE = 0.3189
MLPRegression RMSE = 0.3898                     MLPRegression RMSE = 0.4127
AdaBoostRegression RMSE = 0.3899                AdaBoostRegression RMSE = 0.3621
```

rather than relying on individual decision trees. The averaging idea improves the prediction accuracy and controls over-fitting.

5 Hyperparameter selection and model performance

In this section, we use function GridSearchCV() as our method to tune the hyper parameters and find the best ones. It is a brute force way, trying all possible cases in a range. Even though it costs a lot of time, it is quite easy to use. We tune the following 3 hyperparameters of RandomForestRegression: n_estimators, max_depth and max_leaf_nodes. The best parameters among our range is shown below.

- n_estimators: 190 in range(50, 200, 10)
- max_depth: 15 in range(5, 20)
- max_leaf_nodes: 500 in [5, 50, 500, 5000]
- original RMSE: 0.3664
- best RMSE: 0.3539

Due to the tight time, our range is small and the number of hyperparameters is small. Therefore, the best result may be locally optimal, but not globally optimal. No matter whether the result is optimal, it truly gives us a good improvement on RMSE.

6 PySpark implementation (optional)

We do the PySpark implementation in Feature Engineering. Spark is a distributed processing method, which is an expert at tackling large-data-size problems.

In our project, we use pyspark to accelerate the preprocessing part(Feature Engineering), improving the efficiency. First, we set the pyspark environment and use SparkSession.builder.getOrCreate() to create the spark. Then we use spark to get our training data and testing data.

```
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable
spark = SparkSession.builder.getOrCreate()
training_set = spark.read.csv(xxx)
testing_set = spark.read.csv(xxx)
```

Our implementation mostly use the udf function from pyspark.sql.functions. The general pyspark implementation is shown below in the template.

```
@udf(returnType = xxx)
def functionTemplate():
  xxxxx
```

We have 20000+ rows in the training data, which means doing operations on the column is slow. We first write the basic column operation function, then we need to convert them to the user define function in pyspark. The most easy way is to use "@" in python, we only need to add "@udf" above our written function. Every time we call the function, it uses spark to accelerate.