CS150A Database
Course Project

Yuhan Cao
ID: 2020533165
caoyh1@shanghaitech.edu.cn

Zhenbang Li
ID: 2019531055
lizhb@shanghaitech.edu.cn

Introduction

This project is intended to predict student performance on mathematical problems from logs of student interaction with Intelligent Tutoring Systems. This task presents interesting technical challenges, has practical importance, and is scientifically interesting. In this project, we follow the 5 suggested steps for building a practical machine learning system and implemented with ***Pyspark***, which is also the structure title of this report.

1 Explore the dataset

The training dataset is composed of 19 features (Or 20, if consider *Problem Hierarchy* as a combination of unit and section.), which can be classified into two groups:
1. Numerical features:
Problem View, First Transaction Time, Correct Transaction Time, Step End Time, Step Duration (sec), Correct Step Duration (sec), Error Step Duration (sec), Correct First Attempt, Incorrects, Hints, Corrects, KC(Default), Opportunity(Default).
2. Category features:
Anon Student Id, Problem Hierarchy, Problem Name, Step Name.
After the adequate conversion, the types of the features are:

Row	Anon Student Id	Problem Hierarchy	Problem Name	Problem View
int64	object	object	object	int64
Step Name	Step Start Time	First Transaction Time	Correct Transaction Time	Step End Time
object	object	object	object	object
Step Duration	Correct Step Duration	Error Step Duration	Correct First Attempt Time	Incorrects
float64	float64	float64	int64	int64
Hints	Corrects	KC(Default)	Opportunity(Default)	NaN
int64	int64	object	object	NaN

Fig1. Data types of the features

As shown below, Fig2 is a visualization of the distribution of major features.

However, the testing dataset contains only the 6 category features: *Anon Student Id*, *Problem Hierarchy*, *Problem Name*, *Problem Unit*, *Problem Section*, *Step Name*, which should be emphasised in our training dataset. The testing dataset have some invalid and missing entries that should be cleaned in 1140 rows.

2 Data cleaning

First, we operate on cleaning the invalid values and out-liners in the training dataset. The target of prediction is the Correct First Attempt (CFA). So instead of evenly considering the relations between

the components, we should focus on the relations between each other component and CFA. What's more, there are considerable amount of missing of corresponding CFA, in which case we take the mean CFA of the other same-type elements as the complement. For these category components, we take a modified one-hot encoding. In this way, we generate the corresponding *feature-CFA* of the *Anon Student Id, Problem Name*, *Step Name*, which together with other components construct our training dataset.

3 Feature engineering

As mentioned in the above section, we add the *feature-CFA* relations as new features. To achieve this relation feature, first we should group the dataframe by the original target feature, then filter out those entries without the correct first attempt mark equals 1. For those missing values (entries with correct first attempt = 0), we will take the mean of the other *target feature-CFA* values.
In this way, we quantize a distinct relation between each feature and CFA, reduce the information loss cost by the correlation between features. In our test, these additional features improve the accuracy of prediction, especially for the tree based models. We choose to drop some features like *Correct First Attempt Time(sec)* in the original training data for training, because when we tried to include them in training, the accuracy decreased. This may came from their relative low correlation with other features. What's more, as discussed in the dataset exploration section, these features do not exist in the testing dataset, which makes their weight quite low in contribution of the CFA outcomes.

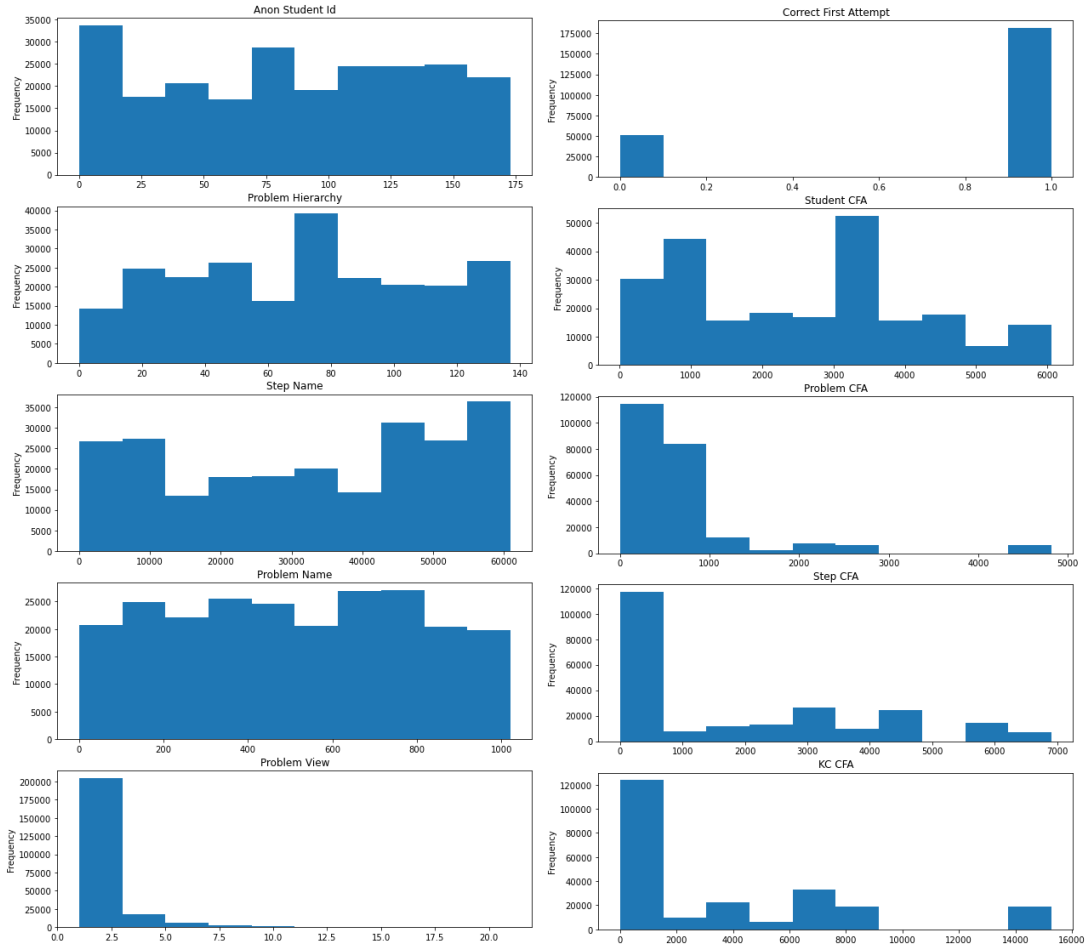

Fig2. Distribution of cleaned and encoded features

4 Learning algorithm

We perform 6 algorithms, including both tree based ones and neural network based ones:

(1) Decision Tree: We choose the naive decision tree as a baseline.
(2) Random Forest: We choose it as a baseline for lightGBM tuning.
(3) AdaBoost Regressor: AdaBoost is adaptive in the sense that subsequent weak learners are tweaked in favor of those instances misclassified by previous classifiers. In some problems it can be less susceptible to the overfitting problem than other learning algorithms.
(4) Logistic Regression: We choose the logistic regression as a baseline and comparison for the tuning of LSTM.
(5) LSTM: Long short-term memory networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series, which is suitable for our case.
(6) LightGBM: Gradient boosting based models have the first tier performance among the tree based classes. LightGBM is a efficient one and relatively easy to tun. Its leaf-wise natural enables faster speed and make it can be efficiently tuned with operations on the depth and restrictions on the sub-trees.

Considering the feature number and size of training set (23744) and testing set (1140), the over-fitting can become the major potential problem for neural network based methods. So we also focus on the tree based algorithm, especially the lightGBM, which turns out to be good after tuning the hyperparameters.

5 Hyperparameter selection and model performance

After training the models on their original hyperparameters and compare the results, we chose the models that have the most potential. We mainly tuned 2 models: LSTM and LightGBM.
(1) LSTM
Our LSTM is implemented based on *keras* and have 3 layer, batch size 32 and epoch 10, the tuning of hyperparameters does not have a satifying outcome. The over-fitting may be the major problem since the dimension of the features is limited.
(2) LightGBM
For the LightGBM, we increase the n-estimator (number of boosted trees to fit) to 1000, which is a common adequate technique according to our background research. Correspondingly, we decrease the n-eaves (maximum tree leaves for base learners), considering the uneven distribution of features in the training dataset. In addtion, to reduce over-fitting, we add the sub samples which can help specify the percentage of rows used per tree building iteration. That means some rows will be randomly selected for fitting each learner (tree). The ratio and frequency of sub samples are tuned in range to find the best result. Fig3 is a table of the major hyperparameters of our LightGBM after tuning.

max-depth	num-leaves	learning-rate	n-estimators	min-child-samples	subsample
5	20	0.1	1000	20	0.85
subsample-freq	boost-from-average	reg-lambda	verbose	boosting-type	n-jobs
1	false	0.1	-1	gbdt	-1

Fig3. Chosen hyperparameters of lightGBM

The final result of our models:

Model	RMSE
Decision Tree	0.5126039
Random Forest	0.4165785
AdaBoost Regressor	0.4276248
Logistic Regression	0.4198125
LSTM	0.4227168
LSTM (tuned)	0.4121097
LightGBM	0.4102616
LightGBM (tuned)	0.3970613
Best RMSE	0.3970613

Fig4. Performance of the models

6 PySpark implementation (optional)

In the data cleaning and feature engineering process, we need to compute the mean value of the *feature-CFA* for complement purpose. We operate a SparkSession, write a *getmean* function and create SQL views and queries for computing the mean of each *feature-CFA* in the process cleaning the train dataset.