Individual Report on XGBoost

April 10, 2022

1 The Model: XGBoost

The task is to predict whether a credit card transaction is fraudulent or not. This is a binary classification task.

Kaggle already split the data into `train_identity`, `train_transaction`, `test_identity` and `test_transaction`. We decided not to use `train_identity` to train our model, only `train_transaction`. The details of data preparation are in the group solution, as we processed the data together and used the same cleaned datasets for individual modelling efforts. In summary, we removed all variables with more than 10% of missing data, imputed categorical missing values using mode and numerical missing values using median.

I chose to use the model XGBoost, which stands for "Extreme Gradient Boosting". Boosting is an ensemble method that combines several weak learners (decision trees in this case) into a strong learner. Models are trained in succession, with each new model being trained to correct the errors made by the previous ones. In the Gradient Boosting approach, new models are trained to predict the residual errors of previous models. The final prediction is a weighted sum of the predictions of the decision trees.

XGBoost has many hyperparameters. I decided to tune a selection of them: `learning_rate`, `n_estimators`, `max_depth`, `min_child_weight`, `gamma`, `colsample_bytree`, `subsample`, `reg_alpha`, `reg_lambda`.

[1]:
```
import numpy as np
import pandas as pd
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.metrics import balanced_accuracy_score
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
df = pd.read_csv('~/Desktop/ID5059P2/train_transaction_cleaned.csv')
df_test = pd.read_csv('~/Desktop/ID5059P2/test_transaction_cleaned.csv')
```

I split the dataset into the covariates and the response, one-hot encoded the covariates (because some are categorical), then used `train_test_split` to split the dataset into 80% training and 20% testing data.

[3]:
```
X = df.drop("isFraud", axis=1)
X_dummified = pd.get_dummies(X)
y = df["isFraud"]
X_train, X_test, y_train, y_test = train_test_split(X_dummified, y, test_size =␣
↪0.2, random_state=3200)
```

2 Fitting Process

I will tune the chosen hyperparameters using GridSearchCV, which does a grid search over a specified parameter grid, as well as k-fold cross-validation. I chose some initial values for the hyperparameters (see below). I chose `balanced_accuracy` scoring method because we have an unbalanced dataset (96.5% of training dataset have `isFraud=0`). This method uses the average recall obtained on each class. Recall is the proportion of true positives correctly predicted for each class (`isFraud=0` and `isFraud=1`), and balanced accuracy is the mean of these.

Note: The full code is only shown for the first grid search. The hyperparameter grids for other iterations and `GridSearchCV` results are shown.

The first two parameters I tuned were `learning_rate` and `n_estimators`.

When each tree is added to correct the residual errors of the predictions made by the model, `learning_rate` applies a weighting factor to the corrections made by the new tree. The higher the learning rate, the more conservative the boosting process, and the longer it takes to train the model.

`n_estimators` is the number of trees used in the model.

Since there are so many hyperparameters to tune, I chose to use a high learning rate and low number of estimators to use while tuning the other hyperparameters (to reduce training time). I also tuned the hyperparameters in pairs, also to reduce training time. Once the rest of the hyperparameters have been tuned, I will reduce `learning_rate` and `n_estimators` for the final model to improve performance.

[ ]:
```
param_test1 = {'n_estimators': [50, 200, 500], 'learning_rate': [0.1, 0.3]}

grid_search1 = GridSearchCV(XGBClassifier(max_depth = 5, min_child_weight = 1,␣
↪gamma = 0, subsample = 0.5, colsample_bytree = 0.5, reg_alpha = 0,␣
↪reg_lambda = 0, seed = 3200),
                            param_grid = param_test1,␣
↪scoring='balanced_accuracy', n_jobs=-1, cv=5) # 5-fold CV

grid_result1 = grid_search1.fit(X_train, y_train)

print("Best score: {}, Best parameters: {}".format(grid_result1.best_score_,␣
↪grid_result1.best_params_))
# output: Best score: 0.747162, Best parameters: {'learning_rate': 0.3,␣
↪'n_estimators': 500}
```

The next two hyperparameters I tuned were `max_depth` and `min_child_weight`. `max_depth` is the maximum depth of a decision tree, and `min_child_weight` is the minimum sum of of weights of all observations required in a child node. Both help control overfitting.

```
[ ]: param_test2 = {'max_depth': [3, 5, 7, 9], 'min_child_weight': [1, 3, 5]}
     # output: Best score: 0.778722, obtained using hyperparameters {'max_depth': 9,␣
      ↪'min_child_weight': 1}
```

Since the best score was when `max_depth = 9`, I did another grid search to see if increasing the value would result in a better score, and it did (see output below). However, I chose not to increase the value further as this could result in overfitting.

```
[ ]: param_test3 = {'max_depth': [9, 11, 13]}
     # output: Best score: 0.782392, Best parameters: {'max_depth': 13}
```

`gamma` was the next hyperparameter, and it specifies the minimum loss reduction required to make a further split on a node of the tree. This means that if creating a new node doesn't reduce the amount specified by gamma, then we won't create it.

```
[ ]: param_test4 = {'gamma':[0, 0.1, 0.2, 0.3, 0.4, 0.5]}
     # output: Best score: 0.782638, Best parameters: {'gamma': 0.1}
```

The next two hyperparameters were `colsample_bytree` and `subsample`. `colsample_bytree` is the subsample ratio of columns used when constructing each new tree, and `subsample` is the proportion of observations to be randomly sampled for each tree. Prior to growing trees, a randomly sampled subsample proportion of the training data prior to growing trees (subsampling happens once in every boosting iteration).

```
[ ]: param_test5 = {'subsample':[0.6, 0.7, 0.8, 0.9], 'colsample_bytree':[0.6, 0.7,␣
      ↪0.8, 0.9]}
     # output: Best score:  0.784279, Best parameters: {'colsample_bytree': 0.9,␣
      ↪'subsample': 0.7}
```

Onto XGBoost's two regularisation hyperparameters: `reg_alpha` and `reg_lambda` (both help reduce overfitting). `reg_alpha` is the L1 (Lasso) regularisation term on the weights. As a result of Lasso, some coefficients are set to exactly 0. `reg_lambda` is the L2 (Ridge) regularisation term on weights. As a result of Ridge, magnitude of coefficients of weights is as small as possible. As these values increase, the model becomes more conservative.

```
[ ]: param_test6 = {'reg_alpha':[1e-2, 0.1, 1, 100], 'reg_lambda': [1e-2, 0.1, 1,␣
      ↪100]}
     # output: Best score:  0.789186, Best parameters: {'reg_alpha': 0.1,␣
      ↪'reg_lambda': 0.1}
```

3 Training the Final Model

In summary, the hyperparametes I chose were: `max_depth = 13`, `min_child_weight = 1`, `subsample = 0.7`, `colsample_bytree = 0.9`, `gamma = 0.1`, `reg_alpha = 0.1`, `reg_lambda = 0.1`. Now, I'll set the learning_rate = 0.01 and n_estimators = 5000. Training the final model:

```
[4]: final_model = XGBClassifier(learning_rate = 0.01, n_estimators = 5000,␣
     ↪max_depth = 13, min_child_weight = 1, subsample = 0.7, colsample_bytree = 0.
     ↪9, gamma = 0.1, reg_alpha = 0.1, reg_lambda = 0.1, seed = 3200);
     final_model.fit(X_train, y_train);
```

```
[20:25:51] WARNING: /Users/runner/work/xgboost/xgboost/src/learner.cc:1115:
Starting in XGBoost 1.3.0, the default evaluation metric used with the objective
'binary:logistic' was changed from 'error' to 'logloss'. Explicitly set
eval_metric if you'd like to restore the old behavior.
```

4 Measuring Model Performance

```
[5]: y_pred_train = final_model.predict(X_train)
     training_accuracy = balanced_accuracy_score(y_train, y_pred_train)

     y_pred_test = final_model.predict(X_test)
     generalisation_accuracy = balanced_accuracy_score(y_test, y_pred_test)

     print(training_accuracy, generalisation_accuracy)
```

```
0.9874891297793174 0.8040415687593481
```

The training accuracy score is 98.7%, which is much higher than the generalisation accuracy score (80.4%), meaning the training data was likely overfit.

To improve performance of this model, I would spend more time preparing the data. Instead of removing any variables with more than 10% of the data missing, I would inspect each covariate and select more carefully and take more care in the data imputation stage. In addition, I could perform more grid searches to find better parameters (although this is less likely to result in a significant improvement in model performance, compared to more careful data preparation).

As a team, we had five in-person meetings to discuss this project. I created a group chat and a Teams channel for the team to make communication easier; organised all the meetings; took notes and shared minutes (along with useful code examples I found) on our group's Teams channel; wrote the code for data preparation, and shared it with the group (they agreed with my decisions, and some did further imputation). After everyone uploaded their final code and score onto Teams, I wrote the first draft of the client report and compiled the first draft of the group final solution. At meetings, every team member in attendance raised issues and contributed to problem solving.

I felt Lucy Brook did not contribute sufficiently to the group. When organising meetings, they could not make any of the suggested times or offer alternative timings they were available. Lucy only attended one meeting and did not start work on their individual model until in tothe final week before the deadline. Similarly, until a couple of days before the deadline there was no participation in any group discussion. Lucy did not fully engage with us or contribute equally to the project.