State Farm Machine Learning Engineer Take-Home Assessment (Exercise 06)

August 4, 2020

Phoi-Tack (Charlie) Lew

Introduction

This is a brief report on a fictitious machine learning case study that was provided by State Farm as the first part of the Machine Learning Engineer assessment. The dataset provided consisted of two csv files separated by train and test. The train data contained a binary target whereas the test (also referred to as 'holdout') does not. The latter being a blind test to determine overall AUC score. There are six steps to follow through in this assessment and hence this report divided into six sections addressing the six requirements.

Step 1

Optimize the model: There are areas for improvement for the base model presented. Modify the code to improve the accuracy of the model. Look for opportunities to improve performance including data cleaning/preparation, model selection, train/test split, and hyper-parameter tuning. The model performance will be measured by AUC against the holdout test set.

The original provided Jupyter notebook contained several methods that were used in this study. There were several inconsistencies in the original notebook. There are namely in no particular order:

- The raw original dataset does not contain any date-like columns but there is a date converter function
- The function of removing the dollar sign was applied to several columns/features that were not needed
- The use of regression when the target variable is not continuous but binary

There were several other inconsistencies but they were just minor ones. Nonetheless, a new notebook was created to start the analysis from scratch and named 'ModelingLew.ipynb'. The following bullet points addresses the key methods that were used in this section:

- The categorical variables are columns x1, x54, x82 and x84. Label Encoding was performed on the aforementioned columns. The rest were numerical.
- The columns that required cleaning were x12 and x79, i.e. remove dollar symbol and convert percentage to its 100^{th} division.
- A train-test split of 80-20 was performed on the training data with 80% training and 20% test.

- The selected model chosen is XGBoost which gave an **AUC of 0.94**. I selected 500 estimators with a tree depth of 6. Random Forest was also used but not selected since XGBoost performed significantly better and faster.
- There was consideration in using other models but were not done so. With an AUC score of 0.94, the model is considered to perform exceptionally well. Further improvements will incur a high cost with potentially minor improvements. In the interest of time, this can be revisited
- No rigorous hyper-parameter tuning was performed in the interest of time. Nonetheless, an attempt was made using BayesianSearch which in my experience works better compared to GridSearchCV and RandomizedSearch. Again, this can be revisited another when there is more time and resources. There is a notebook cell that has the BayesianSearch coded.
- Another observation was that training dataset itself is imbalanced but not severely with about 20% of it belonging to Class 1. This could have an impact on AUC performance depending on the feature set present. There are few ways to potentially improve the AUC score
    - Over-sample the minority class via SMOTE
    - Down-sample the majority class
    - Use imbalance-learn module. There is a claim that it works well for imbalanced datasets. I have not tried this here.

Step 2

Prepare model deployment for production: Update your code to meet common production coding standards and best practices. These include modularization, code quality, proper unit testing, and comments/documentation. This should be completed for all parts of Step 1. The code will be evaluated using tooling that evaluates code coverage and code quality. Place unit tests in a folder called "tests" outside of your project code.

This section is incomplete. My knowledge in this area is minimal and I am currently making efforts to gain an improved background on unit testing and code quality.

Step 3

Wrap the model code inside an API: The model must be made callable via API call. The call will pass 1 to N rows of data in JSON format, and, expects a N responses each with a predicted class and probability belonging to the predicted class. Please use port 5001 for your API Endpoint.

Each of the 10,000 rows in the test dataset will be passed through an API call. The call could be a single batch call w/ all 10,000 rows, or 10,000 individual calls. API should be able to handle either case with minimal impact to performance. Note that whether it is a single call or batch call, it should always return an array.

The model was wrapped in Flask in a file name api.py. Simply run

```
python api.py
```

to execute the API. There are several issues here that I noticed and thus the section still needs improvements. The issues for now are:

- Batch curl calls are limited to about 100 observations. Beyond that there is an error complaining that the argument list is too long.
- Performance on the api.py is an issue particularly because a number of operations are being performed to clean the data before making a prediction.
- Not sure how to print out to numbers that are not strings

In order to handle errors that arise due to data that is uncleaned even after the cleaning process, the prediction will return a Class 0 with probably of 2. This will serve as red flag for later analysis as to why it failed.

An './output/' folder was also created to dump the results to a file with a timestamp. The option here is to open and append the results. What I would like to have done is to add more rigor and sophistication whereby some form of checkpointing is done and hence no repeated calculations are performed thus preserving computational resources.

Step 4

*Wrap your API in a Docker image: Create a Dockerfile that builds your API into an image. Write a shell script titled run_api.sh that either runs your image using traditional docker run commands or orchestrates your deployment using Compose, Swarm or Kubernetes (include relevant *.yml config files).*

This section is incomplete. I have wrapped the API in a Docker image and was able to start the docker image server. However, using curl to pass in the JSON input did not return any results as least through the command line. I am unsure what the issue is here but given enough time this problem will be fixed. As requested, a 'run_api.sh' file was created to create the image and start the service.

The next steps if I had more time are pushing the docker image to docker hub and then using perhaps Google Cloud Platform's (GCP) Kubernetes Engine, provision several nodes and deploy my Docker image

Step 5

Optimize your deployment for enterprise production and scalability: Identify opportunities to optimize your deployment for scalability. Consider how your API might handle a large number of calls (thousands per minute). What additional steps/tech could you add to your deployment in

order to make it scalable for enterprise level production. You can incorporate any relevant code (optional), or you can describe your steps in the write-up as part of Step 6.

Before deploying at the enterprise level, the Flask application has to be efficient and most importantly the bugs that were mentioned above (unable to parse long strings) need to be addressed. Where efficiency is mentioned, I am implying the issues such as the cleaning portion of the JSON input in the api.py file. If these are correctly addressed, then one way to test whether my app will be able to handle large volumes of calls is potentially use Loader.io to send hundreds to thousands of calls to my API. I can use this to perhaps test how efficient the API is. In order to scale to enterprise level, as mentioned in the Step 4 above, one can deploy it via Kubernetes engine and provision several nodes with sufficient compute power and memory. This is an area I am still learning and thus my knowledge in this area is limited for now. Nonetheless, these are what comes to mind.