Chaklam Silpasuwanchai, Akraradet Sinsamersuk
AT82.03: Machine Learning

A3: Predicting Car Price

In this assignment, for the sake of simplicity, you will keep using the `Car Price` dataset but treating the problem as classification. We shall implement Logistic Regression we have learned in class, add some more features on top, perform some experiments using ML flow, and lastly deploy your docker to our prepared virtual machine.

Note: You are ENCOURAGED to work with your friends, but DISCOURAGED to blindly copy other's work. Both parties will be given 0.

Note: Comments should be provided sufficiently so we know you understand. Failure to do so can raise suspicion of possible copying/plagiarism.

Note: You will be graded upon (1) documentation, (2) experiment, (3) implementation.

Note: This is a two-weeks assignment, but start early.

Deliverables: The GitHub link containing the jupyter notebook, a README.md of the github, and the folder of your web application called 'app'.

Task 1. Classification - Based on `02 - Multinomial Logistic Regression.ipynb`, modify `LogisticRegression()` class as follows:

- First, make sure you used the preprocessed version of the dataset that you have done in A1/A2. Following the preprocessed version, convert the label `selling price` into discrete variable by simply putting the price in a bucket of 0, 1, 2, 3, which will result in a 4-class classification problem. One possible function is to use `pd.cut()`, but feel free to use whatever methods to achieve this.
- In sklearn, there is a handy function called `classification report`. Here, we shall learn how to obtain this report from scratch so we can fully understand classification metrics. First, add a function `accuracy` which returns a single number as

$$\text{accuracy} = \frac{\text{correct predictions}}{\text{all predictions}}$$

- Add functions `precision`, `recall`, `f1-score` which calculates the score for each class. Note the equation for each metric for an arbitrary class c as follows:

$$\text{precision}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c}$$

$$\text{recall}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FN}_c}$$

$$\text{f1}_c = \frac{2 * \text{precision}_c * \text{recall}_c}{\text{precision}_c + \text{recall}_c}$$

  where TP, TN, FN, FP are True Positives, True Negatives, False Negatives and False Positives respectively.
- Add functions `macro precision`, `macro recall` and `macro f1`. Macro averaging simply averages the precision, recall, and f1 across all classes. For example, macro precision across 4 classes is simply

$$\text{macro precision} = \frac{\text{precision}_0 + \text{precision}_1 + \text{precision}_2 + \text{precision}_3}{4}$$

- Add functions `weighted precision`, `weighted recall` and `weighted f1`. Weighted averaging is almost the same as macro averaging but instead add a weighting factor in front of each class. This is particularly useful when your dataset class is imbalanced. Let's say your dataset has only 20% of class 0, 30% of class 1, and 20% for class 2, and remaining to class 3. Then the weighted precision for 4 classes is as follows:

$$\text{weighted precision} = 0.2 * \text{precision}_0 + 0.3 * \text{precision}_1 + 0.2 * \text{precision}_2 + 0.3 * \text{precision}_3$$

- Last, try to run scikit-learn `classification report` and compare with your implementations. You can try with any mockup data. Confirm both implementations are close.
- Just a brief question here: what does `support` in the classification report means?

Task 2. Ridge Logistic Regression - Notice that our Logistic Regression does not have option for imposing the ridge or L_2 penalty. Please implement this penalty onto the code and allows users to choose whether to use the penalty or not. The loss function of Ridge Logistic Regression is as follows:

$$J(\theta) = -\sum_{i=1}^{m} y^{(i)} \log(h^{(i)}) + \lambda \sum_{j=1}^{n} \boldsymbol{\theta}_j^2$$

Hint: You can simply copy some of the code at `Regularization.ipynb`.

Task 3. Deployment

In the last assignment, you have been logging your experiment locally on your personal machine which is fine for learning. However, to feel the benefit of using MLflow, you will now log your experiment into the server. If you use this tool correctly, it could replace the report-writing process. But we won't go that far in this class.

For this task, these are your objectives.

1) Log your experiment to the CSIM mlflow server.
2) Deploy the Model using 'Models' module of the MLflow.
3) For the final touch, set up GitHub for CI/CD.

Objective 1: Log experiment on the server.

This task is super easy. Change the `tracking_uri` to `https://mlflow.cs.ait.ac.th/`. Set the experiment name as `<student_ID>-a3`. Don't log the dataset. Save the model.

Objective 2: Deploy the Model.

This is also not that difficult. Study this MLflow - model registry. Only the 'concept' and 'UI Workflow'. Find your best model among the runs. Register your best model to the 'Models'. Name your model `<student_ID>-a3-model`. Make sure the model is at the `staging`.

Objective 3: CI/CD.

This is probably the most difficult part of the entire assignment. Wow, intimidating. Let's start with, what is CI/CD?

CI/CD

CI/CD stands for **C**ontinuous **I**ntegration and **C**ontinuous **D**elivery/**D**eployment. As the name suggested, it is the idea of performing integration and delivery/deployment continuously. Get it? No? Ok.

In modern software development, we want to deliver our software as soon as possible. Imagine that you are working in a company with a fairly big team of developers. Each person works on the same software but with different features. When one feature is developed, you will have to `integrate` the feature to the main software. Once it is integrated and tested, you want to roll out the change right away.

The common problem of software development in general is bugs. Every time you write a new line of code, there is a chance that you create a bug. The longer the code, the higher the number of bugs. When you deploy your software, in this case, in the format of a web application, you probably do not want your software to have bugs (unless you are a maniac). You want to test your software before delivering it. And when you find bugs, you stop the delivery and fix those bugs. At the early stage when your software is simple, and has a few functions/buttons/pages, it is practical to perform the testing manually. However, when your software grows, performing the testing of every function/button/page becomes impossible. So to achieve CI/CD, one thing you will also have to perform is automated testing. For every piece of code

you write, you have to write a test for it. This is a very effective developing framework/practice such that it has its one name which is Test Driven Development (`TDD`) and Behavioral Driven Development (`BDD`).

In summary, what we want to do is constantly integrate, test, and deploy (if the test passes).

TO DO

1) Create an automated testing script. For this assignment, only unit tests. For the Dash project, you can study this page. In the real project, the unit tests are supposed to test every test case on every function in the project. However, just for the sake of your mental health, we only asked you to write the unit tests for your model. Write two unit test functions that test (1) The model takes the expected input and (2) the output of the model has the expected shape.
2) Create GitHub action that once you push a commit, it will evoke the testing script.
3) Update the GitHub action such that the deployment process is automated if the previous testing is passed.

As always, the example Dash Project in the GitHub repository contains an example that you can follow (if you use the Dash framework).

Good luck :-)