TBAuctions. A simple API to build predictions across an XGBoost model.

This PoC only presents the necessary elements that we will use in a real Big Data project. The elements would have the same utilities, but of course, in this PoC, they will not have the same computer power that we can expect to use in a final project.

Here, I defined three containers. The most important is the container to have the model; I will use it to calculate the predictions. In addition, we have a PostgreSQL in which I record the predictions. I also added a second API to give access to the database.

Of course, I added some security to APIs to be sure that only authorized people can use them.

To simplify the test of the API, I added three Jupyter notebooks, in which we can access the APIs.

model_api

This container contains the model and is used to calculate the predictions. In this case, a Docker container could not be loaded with a model, so we defined two functions: one that trains and saves the model inside the container and a second that uses the model, previously trained, to make the predictions.

The "app.py" file defines both endpoints. This file only contains the endpoint and the security level. The code that trains the model or makes the predictions are in the "predictions.py" file.

In the "security.py" file, we have the functions to really define the security and a "FAKE" database, in which we have the data of two users, and only one can have access to the API.

The password is encrypted to ensure that nobody can read it. But the most adequate step will be to define a real database with one table in which we saved the data so that the customer can have access. As we can see in the notebooks the password used is "secret".

The folder "utils" has the parameters that we need to use to work with the containers without problems. Before loading this project in a Docker Container environment, it is necessary to modify the IP parameter contained in the "utils/network.py" file with the IP of the system in

which we will execute the PoC.

Necessary updates to use in a real environment.

As I mentioned before, a real database would be necessary to load the users' data so that they can access the APIs. This would increase security.

A second element that would need to be updated is where we will save the model. In this example, I saved the model inside the container, which generates the additional problem that every time I reload the project, I will need to retrain the model. The most adequate option, which I consider outside of this PoC, is to load the model in a S3 bucket. In the end, the function to train the model will do the same, but the unique difference is that the model is saved outside the container to ensure that possible duplicates of the container can access the same model.

Of course, in this PoC, I didn't need to evaluate how many customers could connect to the system simultaneously. For this reason, using only one implementation of the container is enough. In real life, we will need to use a system to balance the access to the set of containers we will use regarding the number of expected customers. This system will need to start or close containers in real-time. A Terraform environment could help us resolve this problem.

tba_database

This docker is a PostgreSQL implementation. I added this database because I consider that it is imprescindible to load the predictions that we did. We will need this information to can evaluate the quality of our model.

In the real environment, we won't need to define this database because we will use the database defined in the project. It would be interesting to evaluate the idea of using a linear database because, in this structure, we will save the predictions immediately after calculating them.

In our PoC, I defined only a unique table with the data used to train the model, which we will need to use to calculate the predictions. I also added three additional fields. The predictions themselves, the version of the model used, and a timestamp to know when we made the prediction.

database_api

After loading the predictions in the database, we will need to access this data to evaluate the quality of our project.

Of course, we expected that this API was defined previously in the project, but I added it in this PoC because it would be necessary to consider the importance of access to the predictions.

I added only one endpoint that recovers the last "N" predictions, but any other approach would be necessary to add to the final version of the project.

The container itself is a copy of the "model_api" container, with the same level of security, but which its own database, to separate the permissions. Of course, as before, we will need to update the file "utils/network.py" by adding the correct IP.

NoteBooks.

This folder contains the three notebooks developed to evaluate how the APIs work. We only need to update the system's IP to ensure that we can connect to the containers. And, of course, we must execute in the correct order.

The first is the notebook "01. Train Model.ipynb," which uses the file "features.csv" to train the model as the]original notebook. This file is located in the folder "Examples."

After training the model, we can use the notebook "02. Get Predictions.ipynb" to calculate some predictions. In this case, I used the same file "features.csv". It is not the most correct, but it is my unique example.

Inside the notebook, we can decide whether to calculate the prediction of only one bid or a set of bids.

The last notebook, "03. Last Predictions.ipynb," only recovers some data from the database; the system doesn't calculate new predictions; it only recovers the data saved in the database.

How to prove this PoC.

The system is easy to use; we only need a computer with a Docker Container environment. Unzip the file into a folder, access the folder in a terminal, and execute the instruction "docker compose up—-build."

After some seconds, the system appears to be working, and using Jupyter, we will access the notebook to execute them.