upGrad

Automating The Model Building Pipeline

- There are two pipelines present in the ml_pipeline category. These are the model building pipeline and inference pipeline.
- Inside the model building pipeline, you learnt that the different tasks (operators) present in the model pipeline are mainly the same as the tasks of the data pipeline, except for one task. The reason for having similar tasks is that the model pipeline builds the machine learning model on the processed data (i.e., data on which various transformations and feature engineering are applied), which comes from the data pipeline. The new task which has been added to the model pipeline is op_model_training_with_tuning .
- Then, you started working on the 'pipeline_model_building.py' file, where you did the following:
    - Imported the required libraries and called the module_from_file function to load the utils.py file.
    - Defined the name of the experiment in a variable called 'experiment_name'.
    - Called the mlflow.set_tracking_uri function to tell MLflow the location/URL of the server where it is running or has been initiated. The URL of this server is stored in the 'mlflow_tracking_uri' variable.
    - Note: The variables such as 'experiment_name' and 'mflow_tracking_uri' along with other variables are defined in the file 'constants_model_building.py'
    - Defined a try-except block, in which you called the logger.info method and mlflow.create_experiment function, respectively. The logger.info method logs a simple message stating that the experiment has been created, whereas the mlflow.create_experiment function creates an experiment with a name passed as an argument. Recall that in your case, you created an experiment with the name defined by the 'experiment_name' variable (which you had defined earlier). However, if there is any exception, then the code goes to the 'else' block.
    - Called the mlflow.set_experiment function to set the experiment to 'active' in the environment.
    - Now, what is the difference between the mlflow.set_experiment and mlflow.create_experiment functions? The mlflow.create_experiment function creates an experiment and returns its ID. Let's say you created an experiment named 'new_experiment'. Now, to create a run under this experiment, you need to pass its ID to the mlflow.start_run function (which you will learn shortly). Now, the mlflow.set_experiment basically sets an experiment to active, and if no argument is passed to the mlflow.start_run function, then all the runs are launched under this experiment. If the experiment does not exist, then the mlflow.set_experiment function first creates it and then sets it to active.
    - Defined a DAG with the same parameters as the data pipeline except for the 'tag' argument. This time, the value that you passed to the tag parameter was 'ml_pipeline'.

- Defined the op_model_training_with_tuning task. The different parameters passed to this function are as follows:
  - task_id: 'Model_Training_hpTuning'
  - python_callable: get_train_model_hptune() function, which is defined in utils.py
  - op_kwargs: {'db_path': db_path, 'db_file_name': db_file_name, 'drfit_db_name':drfit_db_name}
  - dag: the DAG object that you created

get_train_model_hptune() function -

- This function takes three arguments – db_path, db_file_name and drfit_db_name – and performs hyperparameter tuning for the lightbgm (lgbm) model.
- The function starts by establishing a connection with the database. It then reads the X (features) and y (target variable) from the X and y tables respectively. It then calls the train_test_split function over X and y, and divides them into X_train, X_test, y_train and y_test respectively.
- The function then creates indices of categories and stores them in the 'index_of_categories' variable. This is done so that the lgbm can perform the conversion of category columns into numerical ones.
- Then, it calls the StratifiedKFold function, which first generates the train/test indices and then uses these indices to split X and y into train and test dataframes, respectively.
- Then, it defines various grid (learning_rate, objective etc) and model (objective, num_boost_round etc) parameters.
- After defining the grid and model parameters, it creates an LGBMClassifier object and sets model parameters using the set_params function.
- Then, it calls a BayesSearchCV function and creates a search object. Using this search object, it tries to fit on the estimator (mode passed) with randomly drawn parameters and stores the best model and the best score, respectively.
- Then, it runs a loop over the grid parameters dictionary (defined earlier) and prints their values for the best model.
- The **get_train_model_hptune()** function uses the **mlflow.start_run** function to start a new run under a particular experiment. If no experiment name is passed to the **mlflow.start_run** function as an argument, it uses the experiment which has been set active by the **mlflow.set_experiment** function.
- Then, it calls the **predict** method over the best_model (estimator) and stores the result in the 'y_pred' variable.
- Then, it calls the **mlflow.sklearn.log_model** function to log the model at a specific path defined by the argument **artifact_path**.
- Then, it calls the **mlflow.log_param** function over the grid parameters dictionary and logs the hyperparameters.

- Then, it calculates different metrics, such as accuracy, precision, recall etc. and logs them using the **mlflow.log_metric** function.
- All the parameters, metrics and artifacts that **get_train_model_hptune()** logs can be checked on the MLflow dashboard.

Automating The Model Inference Pipeline

The model inference pipeline extracts the trained model from the path given in the 'ml_flow_model_path' variable (present in the constants file). Once loaded, the pipeline will use the trained model to serve the predictions through the operator **op_predict_data**, which calls the **get_predict()** function to generate the predictions for the batch input data.

First, you need to load the database and check the process flags for Prediction. If it is 1(ON), you took the following steps:

- Called the **mlflow.set_tracking_uri()** function to inform MLflow of the location/URL of the server where it is running or has been initiated
- The URL of this server is stored in the ‘mlflow_tracking_uri’ variable.
- Note: The variables such as ‘experiment_name’ and ‘mflow_tracking_uri’, along with other variables, are defined in the file ‘constants_model_building.py’.
- Loaded the model using **mlflow.sklearn.load_model()** and the path given in the ‘ml_flow_model_path’ variable (present in the constants file)
- Loaded the test data and apply the **.predict()** and **.predict_proba()** methods to generate the predictions and their probabilities, respectively
- Mapped their index with their respective **msno** to accurately track each member ID and their predictions. This is done through column **index_msno_mapping** from the test data.
- Once all the columns were merged, you stored the final predictions in the database for further analysis

Once triggered, you can see the logs of each task defined in the pipeline. In the final predictions, you can also see the probability scores of a user getting churned or not in two separate columns. This result can then be passed on to the marketing team. The team can then analyse and perform the necessary resolutions to retain the user.

upGrad

Automating A Model Monitoring Pipeline

There are four pipelines in the production environment:

- Data_End2End_Processing
- Model_Building_pipeline
- Inference
- Drift_pipeline (Model Monitoring pipeline)

Broadly, the function of the drift pipeline is to check the level of drift between the incoming (new) data and the reference (old) data. Based on the level of the drift, this drift pipeline performs the following actions:

- If the level of the drift is below the primary threshold, then no action is needed, considering that the model will be able to handle the minor deviations in the incoming data.
- If the level of the drift is between the primary and secondary thresholds, the model will not be able to accurately predict on the new data. In such a situation, the drift pipeline should train the model again on the new data to learn the recent data distribution.
- If the level of the drift is between the secondary and tertiary thresholds, the model's performance will worsen further. However, it is still not advisable to throw away the deployed model. Rather than training a new model, you can tune the model on the new data to learn the data distribution better.
- If the level of the drift is beyond the tertiary threshold, then the deviation in the data is beyond the understanding of the current model. In this case, you have to go back to the development environment and perform the model experimentation to find another model that fits on the new data.

NOTE: The levels of primary, secondary and tertiary thresholds vary as per the requirements of the business. However, to calculate the level of drift, you need to load both the old and the new data to understand the delta/change in their distribution. Therefore, the drift pipeline broadly covers all the tasks defined in the data and model-building pipeline.

Later, you worked on the 'pipeline_drift_monitoring.py' file, where you did the following:

- You imported the required libraries and called the module_from_file function to load the utils.py file.
- You defined the name of the experiment in a variable called 'experiment_name'.
- You called the mlflow.set_tracking_uri function to tell MLflow the location/URL of the server where it is running or has been initiated. The URL of this server is stored in the 'mlflow_tracking_uri' variable.

Note: The variables such as ‘experiment_name’ and ‘mflow_tracking_uri’ along with other variables are defined in the file ‘constants_drift.py’.

- You defined a try-except block, in which you called the logger.info method and the mlflow.create_experiment function. The logger.info method logs a simple message stating that the experiment has been created, whereas the mlflow.create_experiment function creates an experiment with a name passed as an argument. Recall that you did the same in the model building pipeline as well. However, if there is any exception, then the code goes to the 'else' block.
- You called the mlflow.set_experiment function to set the experiment to 'active' in the environment.
- You defined a DAG with the same parameters as the data pipeline except for the 'tag' argument. This time, the value that you passed to the tag parameter was ‘ml_pipeline’.
- While defining the task op_reset_processes_flags, you set the parameter Flip to ‘False’. This is done to set the process flags for each and every element of the pipeline to 0(OFF) by default. The tasks in the drift pipeline depend on the drift between the new and old data; therefore, certain tasks will be skipped if the drift level is low.
- You also defined tasks for creating two databases: one for storing all the process flags and the other for storing the features. By defining these two databases, you ensure that the drift pipeline is not dependent on any other pipeline.
- All the other tasks in the pipeline will be the same as in the data and the model building pipeline, except the task of op_get_drift_data.
- You defined this task by passing the following parameters passed to this task:
    - task_id: ‘get_drift’
    - python_callable: get_drift, which is defined in utils.py
    - op_kwargs: {'old_data_directory':old_data_directory, 'new_data_directory':new_data_directory, 'db_path': db_path, 'drfit_db_name':drfit_db_name, 'metric':metric, 'start_date':start_date, 'end_date':end_date }
    - dag: The DAG object that you created

op_get_drift_data function -

- This function takes seven arguments:
    - Two for defining the directories of old and new data: old_data_directory, new_data_directory
    - Two for the databases created earlier: one for the process flags and the other for storing the features
    - One for defining the metric (default = ‘std’) to calculate the drift of transactions and user_logs in the old and new data (These two data sources are selected because of their dynamic nature.)

    - Two for setting the window (start and end dates) of the input data
- The function starts by establishing a connection with the database. It then loads the old and new data of transactions and user_logs. While loading the data, they are compressed, and their date columns are fixed for efficient and fast processing.
- The function then selects all the columns from transactions and user_logs data, which are of int and float data types. The drift level is calculated for these columns only, and all the columns of date type are excluded.
- Once all the numerical columns are extracted, the function calls the get_data_drift for calculating the drift between the old and new data. Let's understand how this works:
    - The function takes multiple arguments to load the old and new data, columns to be included and excluded, metric to calculate the drift and lastly, the database that connected earlier.
    - If the metric chosen is standard deviation ('std), then the function calculates the standard deviation for all the numerical columns in the old and new data. If the metric chosen is the mean, then it calculates the mean for all the same columns. The resulting values are stored in the drift_dict dictionary.
    - Once the defined metrics for old and new data are calculated, the function calculates the absolute difference between the two using the get_change() function.
    - Once the percentage difference of the drift along with the timestamp is calculated, the drift level is stored in the database.
    - Finally, the function returns the average of std_deviation_percentage/mean_deviation_percentage, depending on the metric chosen for calculating the drift.
- Once the drift level between the old and new data of the transactions and user_logs is calculated, the function performs different operations depending on the level of the drift returned.

pd.DataFrame{}

- The function creates a dataframe with two columns: average drift levels of transactions and user_logs.
- The resulting dataframe is then stored in the database for further use.

But how do you decide which tasks should be triggered or not? To decide this, we call in the get_drift_trigger() function.

get_drift_trigger() function

- The function first loads the process flags and the drift dataframe from the database stored earlier.

- To trigger all the tasks in the pipeline, we take the average of both the columns from the drift dataframe. This single value decides which tasks are to be performed and which are to be skipped.
- Based on this average level, you tweak the process flags for each of the tasks in the pipeline to perform the following operations:
    - If the level of the drift is below 10 (primary threshold), then **no action is needed**, considering that the model will be able to handle the minor deviations in the incoming data.
    - If the level of the drift is between 10 and 20 (primary and secondary thresholds), the model will not be able to accurately predict on the new data. In such a situation, the drift pipeline should **train the model again on the new data** to learn the recent data distribution.
    - If the level of the drift is between 20 and 30 (secondary and tertiary thresholds), the model's performance will worsen further. However, it is still not advisable to throw away the deployed model. Rather than training a new model, you can **tune the model on the new data** to learn the data distribution better.
    - If the level of the drift is beyond 30 (tertiary threshold), then the deviation in the data is beyond the understanding of the current model. In this case, you have to go back to the development environment and perform the model experimentation to find another model that fits on the new data.

Creating A Visualisation Dashboard Using Streamlit

Streamlit is an open-source app framework to build web apps for Machine Learning and Data Science tasks. To write an app in streamlit, you need to write a python file simply.Streamlit integrates the python file with the web page/dashboard automatically. All the updates to the python code are reflected on the web page at the same instance.

- Use the following command to install Streamlit:

  pip install mlflow==**1.29.0** pip install streamlit

  NOTE:To run the Steamlit, you need to upgrade the version of mlflow to 1.29.0

- Once the streamlit has been installed, you need to run the python file that you have written for creating the web page. For demonstration, you can run the following command:

  streamlit run /home/dashboard/test.py --server.port **6008**

- As soon as you run the script, as shown above, a local Streamlit server will spin up, and your app will open in a new tab in your default web browser. The app is your canvas, where you'll draw charts, text, widgets, tables, and more.
- To read more on creating widgets, charts, etc., kindly read the documentation for better understanding.

Once you have successfully built the web page for the test.py file, you can also run the main.py(python file for creating the web page of all the pipelines) using the following steps:

NOTE: To run all the pipelines, make sure you have run the following commands to run airflow and mlflow in their respective ports:

- *Run MLflow server on a terminal*

  mlflow server --backend-store-uri='sqlite:///database/mlflow_v01.db' --default-artifact-root="/home/mlruns/" --port=**6006** --host=**0.0.0.0**

- *Run Airflow server on another terminal*

  airflow webserver --port **6007**

- *Once the webserver is connected, run the scheduler on a separate terminal*

  airflow scheduler

- *Once all the servers for MLflow and Airflow is connected, run the following command to open the Streamlit web page*

  streamlit run /home/dashboard/main.py --server.port **6008**