Problem Statement

Build an end-to-end solution that yield scalable & accurate predictions for the given data. The problem statement is split into two categories *product sales forecasting* and *scalability of application.*

1. Product Sales forecasting – Build an ML model to predict the amount of sales for a product based on the features like state, store, sell price, etc.
2. Scalability of application –
   a. Build an efficient solution for hyper parameter optimization for the ML model which uses mean absolute error (MAE) as an objective function.
   b. Implement CI / CD pipeline.
      - Incremental load pipeline to consume new data & check its quality.
      - CI framework for testing whether developed source code is suitable for deployment.
      - Prediction consumption pipeline.
      - Model performance monitoring framework.

Solution Approach:

The above problem can be viewed as ML-Ops (machine learning system operations) problem, as the problem implicitly mimics the stages of ML-Ops like initial data analysis, orchestrated experiment (ML model building), CI framework to test the code before pushing to staging / production, CD pipeline to push the code changes from development stage to production one and lastly continuous performance monitoring of the trained model.

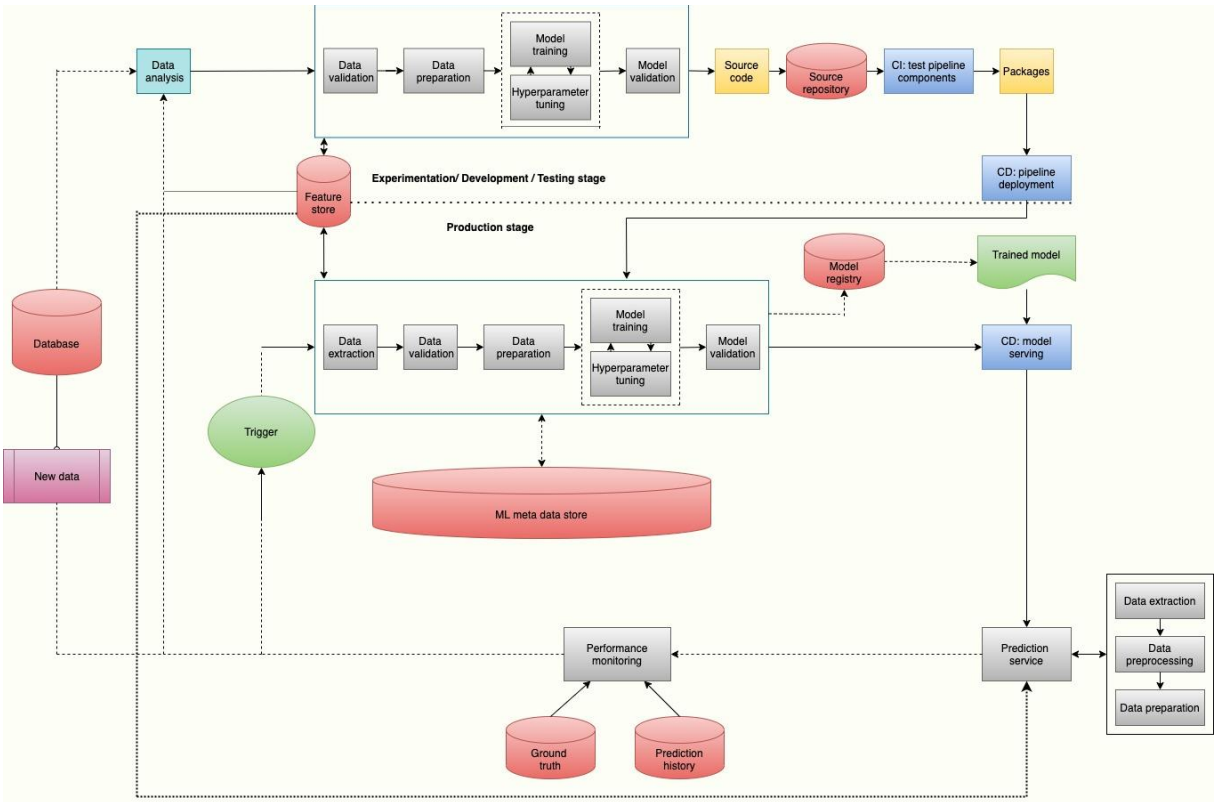


Fig: 1 – ML-Ops architecture

Section 1: Forecasting

The training data set, which is of historical unit sales data has been provided, that has the information of item which includes

- id
- Item id
- Department id
- Category id
- Store id
- State id
- Selling price
- Sales

Except selling price and sales, rest of the information are categorical values. Along with the training data, testing data is also given which contains all information of the item similar to training data except sales value – which needs to be predicted.

ML model building steps:

1. Data Extraction – Given training data is in CSV format which has been imported using pandas. (Utilities/utility.py)

2. Data preprocessing – column id in the given dataset has to be removed as per the given instructions. Cases like checking if there are any nan or improper values has been handled by removing the corresponding rows from the data set. If nan values present in categorical columns it must not be replaced with any other value as it will impact the model performance because of false input data. (Utilities/utility.py)

3. Data Visualization – graphs have been plotted to understand the structure & to get some insights about the data. Data visualization has not been included in the developed project code but in jupyter notebook. (notebook/adidas_task.ipynb)

4. Feature extraction – As categorical columns are in string format which can't be used for model training (must be numerical values), has been encoded and saved in pkl files, so that same encoding format can be used while testing. (Utilities/utility.py)

5. Model Training – for the given regression problem xgboost algorithm has been used as per the given instructions. (Utilities/train.py)

All steps of ML pipeline have been integrated in **execute.py** file.

Section 2: Scalability

2.1 Hyper parameter search

An ML model performance can be improved by tuning the hyperparameters. There are various approaches to obtain best hyper parameter values, grid search and random search are the two commonly used ones. In the given problem description hyper parameters with range of values has been given.

- Learning rate = [0.05, 0.1, 0.15, 0.20, 0.25]
- Max depth = [3, 4, 5, 6]
- Gamma = [0, 1, 2, 5, 10, 15]
- Min child weight = [1, 3, 5, 7]

In total there are 480 different combinations in the search space, for each of these combinations a model has been trained and loss function value (mae) has been calculated. Since training model for 480 different combinations adds time complexity, the search can be done in parallel way. For parallelized approach, pyspark pandas UDF has been used instead of normal UDF. Spark uses executors to process operations in a parallel way which is controlled by the driver. With normal spark UDF, for each executor a python process will be started, & data will be serialized & deserialized between executor and python to process, which leads to impact performance & overhead on spark job. Whereas in pandas UDF or vectorized UDF, Apache arrow is used to transfer data in an efficient way, which drastically improves the performance.

The pandas UDF has been implemented using decorative approach in python, since we need to pass the training and evaluation data to all the nodes along with hyper parameter chunks (by default from spark). One more option is to export the preprocessed training & evaluation data to csv file and importing the same with UDF, but this approach may end up in deadlock sometime when a node doesn't release the lock on a file after importing. (Utilities/training_utility.py)

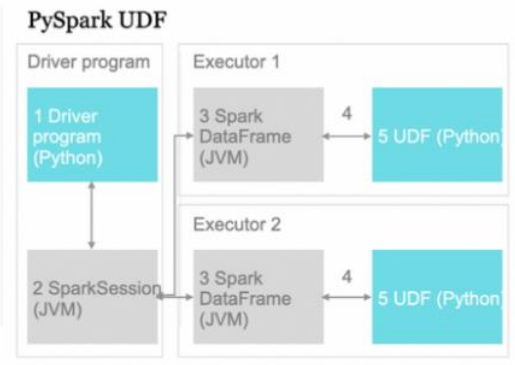


Fig 2: Architecture of pyspark UDF with two worker nodes (executors)

Each of these nodes will return the dataframe of hyperparameters and mae. Then hyperparameter combinations with minimum mae value is calculated, which acts as a

hyperparameter for training the final model. If there are more than one combination who have same minimum mae value, one of them is randomly selected. (Utilities/train.py)

2.2 Continuous prediction for regular timeline

Based on problem description a prediction service has been implemented using bottle framework. The service acts like a server which accepts the prediction request from the client along with data which needs to be predicted. Features will be extracted from the received data in the first step, which will be fed to trained regression model obtained from the registry (meta-data-store) to get prediction. The predicted data will be sent back to the client as a response. (Prediction_service/prediction_service.py)

A client module has been implemented which sends api request to the above prediction service based on predefined time interval. (prediction.py)

2.3 Performance monitoring & Incremental load pipeline

As mentioned before the prediction service uses the model from registry for predicting the client data. But the model being used need to be evaluated time-to-time, because of situations like data-schema skew, data value drift, etc.

An ML pipeline has been implemented along with bottle framework which acts as a server, that accepts request from performance monitoring service & incremental load service to retrain the existing model completely or on top of existing one based on certain conditions. The retrained model will be compared with existing one over a specific evaluation data set, just to make sure retrained one is better than existing one. And the deviation between model's performance will be logged. (ml_pipeline.py)

- The performance monitoring client gathers the historic prediction data for a predefined interval of time and corresponding ground truth data to calculate the deviation between actual value and predicted value. If the deviation is above predefined threshold, information will be sent to data analysts' team & if needed model retraining request will be sent to ML pipeline server. (Performance monitoring/ performance_monitoring.py)
- When new data is added for the next forecasting cycle the existing model has to be retrained, but before retraining data validation has to be done to decide whether to retrain the model or stop pipeline execution. Data validation steps are as follows:
    a. Data schema skew -
        - if the added new data doesn't comply with expected schema **stop** pipeline execution.
    b. Data value drift –
        - Introduction of new categorical data is like introduction of new item, or new store in a state, etc. where existing model's weights can be used to train a new model.
        - Data shift w.r.t selling price or sales for an item. This case new model has to be trained.

So, to tackle the above conditions, whenever a new data is added to the repository the incremental_load.py file will send api request to ML pipeline along with added new data. The pipeline will do data validation steps, based on data validation report either it stops pipeline execution, or a new model will be trained & deployed. (incremental_load.py)

2.4 CI framework

Continuous integration is a practice of integration of code changes from multiple contributors into a single project & also validation of that code before integrating using certain strategies like unit tests, integration tests, model validation tests, etc.

Based on problem description a CI framework has been implemented to test whether the development code should be the production version. Using pytest package in python, unit tests and integration tests has been implemented.

(tests/all test files can be found)

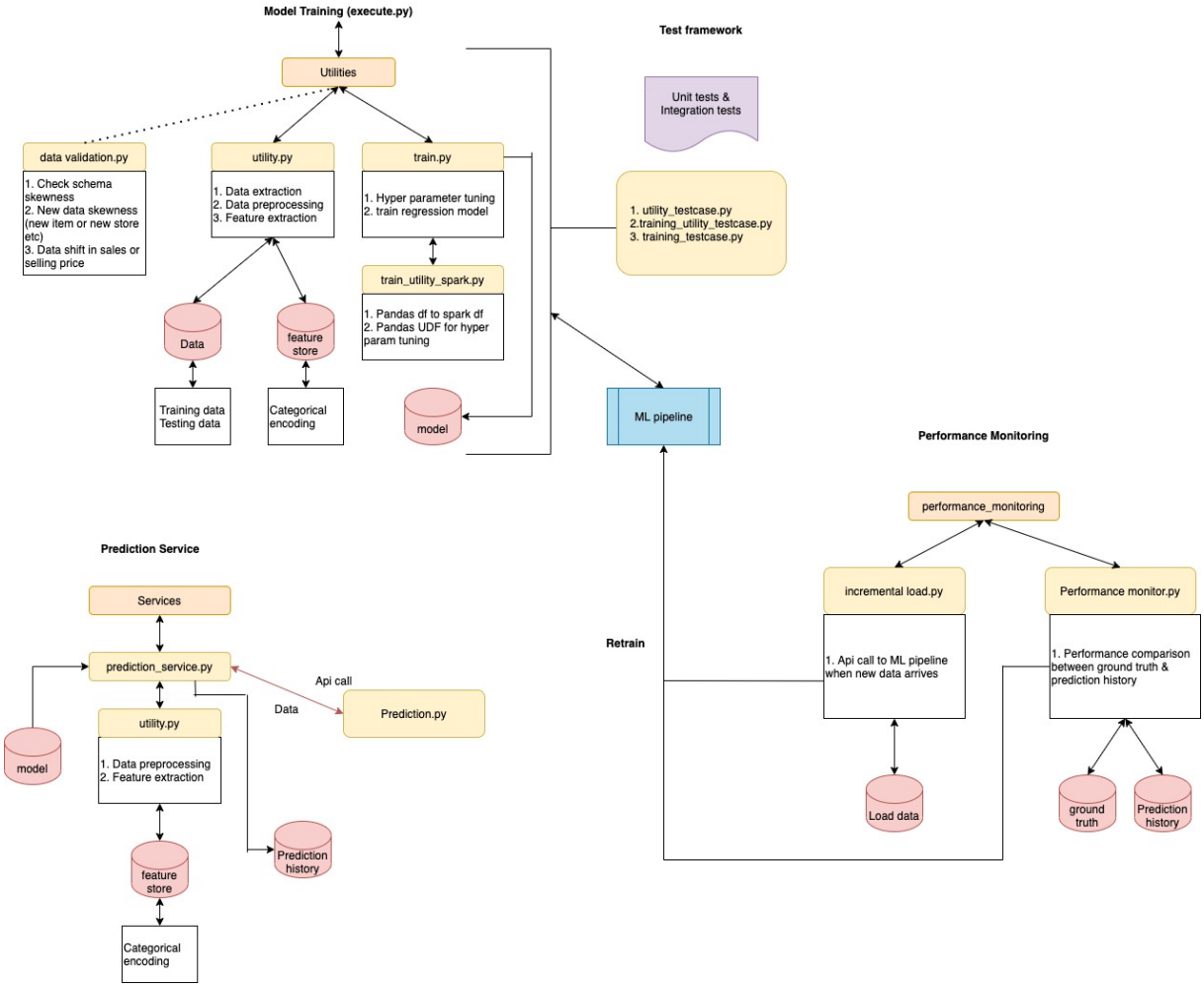
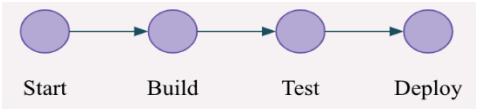


Fig 3: Block diagram of project architecture

2.5 CI – CD pipeline using Jenkins

Jenkins is an open source automation server which allows to build, test & deploy the developed code / software. For this project a Jenkins pipeline has been implemented which triggers when code changes have been pushed to GitHub repository. The pipeline contains 3 stages which are *build, test* & *deliver.*

- In the build stage a docker image of python 3.7 version will be imported, on top of it required python packages will be installed. Then project will be compiled and stashed to a predefined directory.
- In the testing state a docker image of pytest is imported which will be used to run unit tests and integration tests of the code from utility_testcase file. And the test report will be dumped.
- If all the tests are passed, then the compiled and stashed code will be deployed to another repository.



Some insights on Spark configurations:

As mentioned before for parallelization of hyper parameter search, I have used spark's pandas UDF. As spark uses driver – executor or master - slave concept to perform parallelization of process or multiprocessing, different configuration of this has been tested to analyze the performance.

Group by parameter	No of data chunks passed to UDF	Total time taken for execution (sec)
gamma	6	23.74
learning_rate	5	24.25
max_depth	4	26.35
row	480	155.25

Table 1: Hyper parameter search performance using spark with different size of data chunks

No of worker instance	Memory of each worker node	Number of cores assigned for each	Total time taken for execution (sec)
2	1GB	1	38.25
2	1GB	2	28.8
2	2GB	2	27.64
2	2GB	3	23.74
2	3GB	3	25.27
4	1GB	2	27
4	2GB	2	30.23
6	1GB	1	32.56
6	2GB	1	29

Table 2: Spark worker node configuration and its performance

Used machine for this performance testing is Macbook pro with 16GB RAM and i7 processor. From table 1, it can be observed that with different number of chunks, the time taken to perform hyperparameter search varies. Since with parameter 'gamma' that divides the 480 rows of hyper parameter data into 6 chunks provides better result compare to other ones, same configuration has been used for the pipeline.

From table 2, it can be observed that different number of nodes, memory assignment & core assignment has been done. The combination of 2 nodes with 2 GB of memory and 3 core each performs better compare to the rest. In theory with more nodes the performance must be improved, but here its not the case since provided data set is small process time for each node will be less compare to its initialization. One more thing to point out is each node will use only around 50% of the memory assigned to it because rest of the memory will be used by default system call configurations & JVM garbage collections.