

CS5830: Big Data Laboratory

Final Project

Report

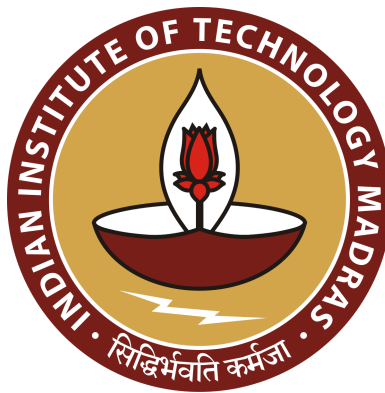
Faculty: Prof. Balaraman Ravindran (ravi@cse.iitm.ac.in)

Guest Faculty: Mr. Sudarsun Santhiappan (sudarsun@gmail.com)

Submitted By: Group 16 - Keerthana, Rinkle, Aravind

Roll Number: ED19B014, NA19B056, CE19B099

Date: 18/05/2024



Indian Institute of Technology Madras

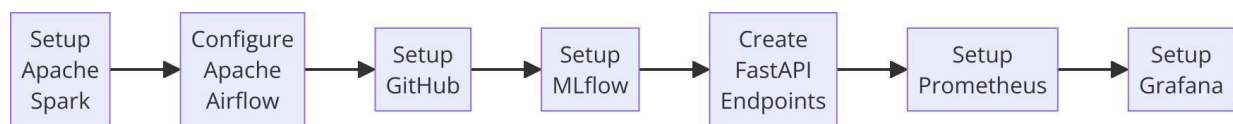
Chennai 600036, India

Repo Link: https://github.com/Keerthana-Senthil/Mlops_project

Problem Statement & Workflow

Our project aims to predict avocado prices using a comprehensive MLOps pipeline. We start with data preprocessing using Apache Spark, which includes automating data acquisition, cleansing, munging, and transformation.

Workflow orchestration is managed with Apache Airflow to streamline and automate the entire process. Detailed project documentation is provided on GitHub, including all source code, models, metrics, reports, and figures, structured with version control using GitHub LFS. Experimentation is meticulously tracked with MLflow, capturing parameters, metrics, and data files. We develop FAST API endpoints for model interaction, instrument these endpoints with Prometheus to monitor metrics, and create a Grafana dashboard for real-time visualization of analytics. Finally, the entire API is Dockerized for seamless deployment, ensuring resource limitations and port mapping are handled efficiently.



Data Set:

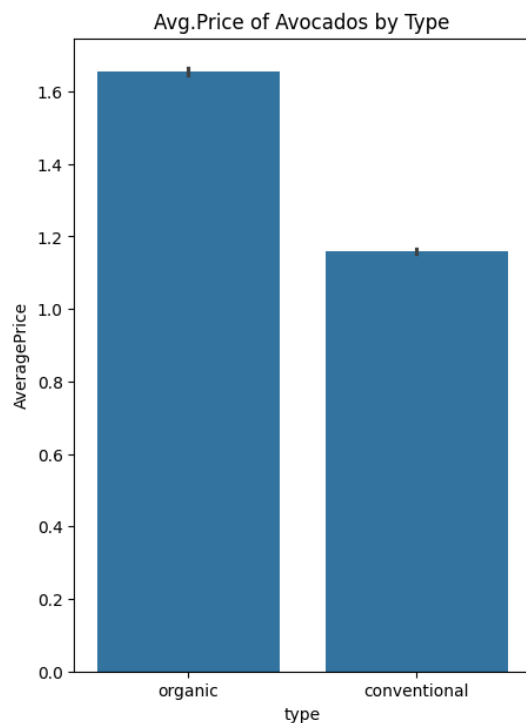
- The Avocado Price Dataset provides data on avocado sales and prices over several years across various regions in the United States. The dataset is publicly available on Kaggle and contains detailed information on different aspects of avocado sales, including the average price, volume, and the type of avocados (conventional or organic).

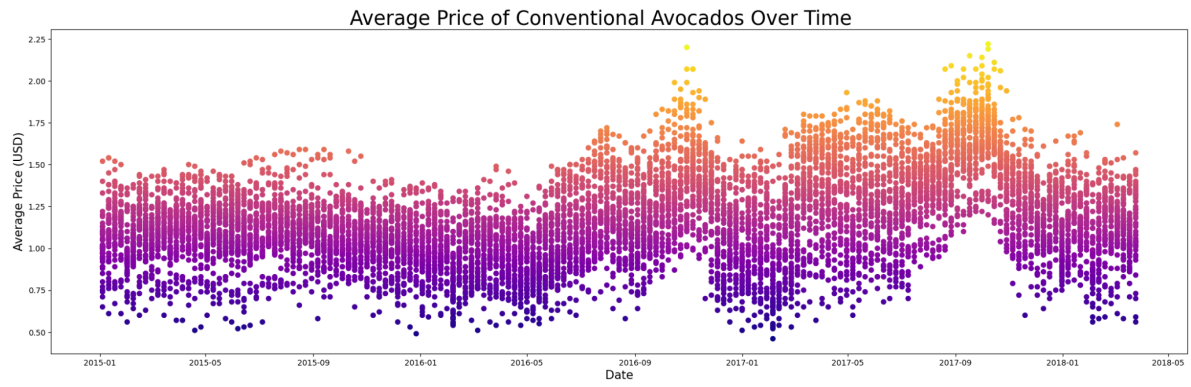
<https://www.kaggle.com/datasets/neuromusic/avocado-pric>

[es/data](#)

- Features:

- Date - The date of the observation
- AveragePrice - the average price of a single avocado
- type - conventional or organic
- year - the year
- Region - the city or region of the observation
- Total Volume - Total number of avocados sold
- 4046 - Total number of avocados with PLU 4046 sold
- 4225 - Total number of avocados with PLU 4225 sold
- 4770 - Total number of avocados with PLU 4770 sold





Apache Spark

Preprocessing function using Apache Spark.

```
from pyspark.sql import SparkSession

spark = (
    SparkSession.builder
    .appName("spark-ml")
    .config("executor.memory", "4g")
    .master("spark://spark:7077")
    .getOrCreate()
)
```

Transformers and Estimators are used to preprocess the data. Transformers convert DataFrames into other DataFrames, while Estimators fit on DataFrames to produce Transformers.

- SQLTransformer: Transforms data using SQL queries.
- Feature Engineering: Log transformation and extracting date components.

```
from pyspark.ml.feature import SQLTransformer

sql_trans = SQLTransformer(
    statement="""
    SELECT
    AveragePrice, type,
    LOG(`4225`+1) AS `LOG 4225`,
    LOG(`4770`+1) AS `LOG 4770`,
    LOG(`Small Bags`+1) AS `LOG Small Bags`,
    LOG(`Large Bags`+1) AS `LOG Large Bags`,
    LOG(`XLarge Bags`+1) AS `LOG XLarge Bags`,
    YEAR(Date)-2000 AS year,
    MONTH(Date) AS month
    FROM __THIS__
    """
)

df_avocado_train_transformed = sql_trans.transform(df_avocado_train)
df_avocado_train_transformed.show(4)
```

Transformations included:

1. VectorAssembler: Combined multiple columns into a single vector column
2. MinMaxScaler: Scaled the month column to a [0, 1] range.
3. StringIndexer: Converted categorical columns into numerical format.
4. Numerical and categorical features were assembled into vectors and scaled as needed.

```
# Load the dataset
data = spark.read.csv('avocado.csv', header=True, inferSchema=True)

# Data preprocessing with Spark
data = data.withColumn('type', data['type'].cast('string'))
data = data.withColumn('Date', data['Date'].cast('date'))
data = data.withColumn('Month', month(data['Date']))
data = data.withColumn('Year', year(data['Date']))

# One-hot encoding for categorical features
data = data.drop('Date')

# Assemble features into a vector
feature_columns = [col for col in data.columns if col not in ['AveragePrice']]
assembler = VectorAssembler(inputCols=feature_columns, outputCol='features')

# Standardize the data
scaler = StandardScaler(inputCol='features', outputCol='scaled_features')

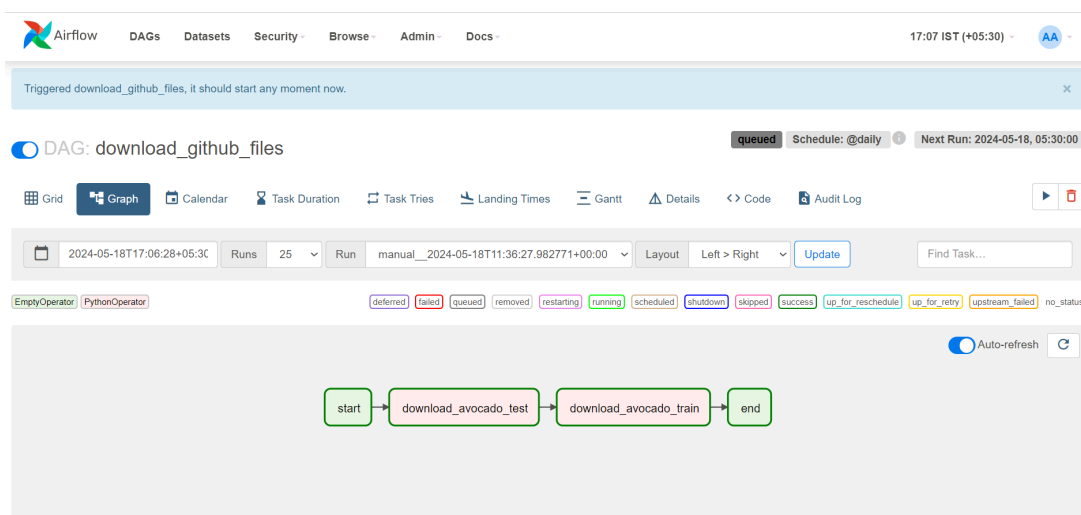
# Define the models
models = {
    "Decision Tree": SparkDecisionTreeRegressor(featuresCol='scaled_features', labelCol='AveragePrice')
}
```

Airflow

Run **airflow standalone**

In the browser at <http://localhost:8080/>, the airflow dag can be run (**dag named download github files**)

DAG visualization - Graph : Using airflow to fetch and download the dataset from the repository.



```
with DAG('download_github_files', default_args=default_args, schedule_interval='@daily', catchup=False) as dag:
    start = DummyOperator(task_id='start', dag=dag)

    download_avocado_test = PythonOperator(
        task_id='download_avocado_test',
        python_callable=download_file,
        op_kwargs={
            'url': 'https://raw.githubusercontent.com/Keerthana-Senthil/Mlops_project/main/dataset/Avocado_test.csv',
            'local_path': '/opt/airflow/dataset/Avocado_test.csv',
        },
    )

    download_avocado_train = PythonOperator(
        task_id='download_avocado_train',
        python_callable=download_file,
        op_kwargs={
            'url': 'https://raw.githubusercontent.com/Keerthana-Senthil/Mlops_project/main/dataset/Avocado_train.csv',
            'local_path': '/opt/airflow/dataset/Avocado_train.csv',
        },
    )

    end = DummyOperator(task_id='end', dag=dag)

start >> download_avocado_test >> download_avocado_train >> end
```

Automation: Reduces manual effort and ensures consistent preprocessing and augmentation of images.

Reproducibility: Provides a repeatable process that can be easily

triggered and monitored.

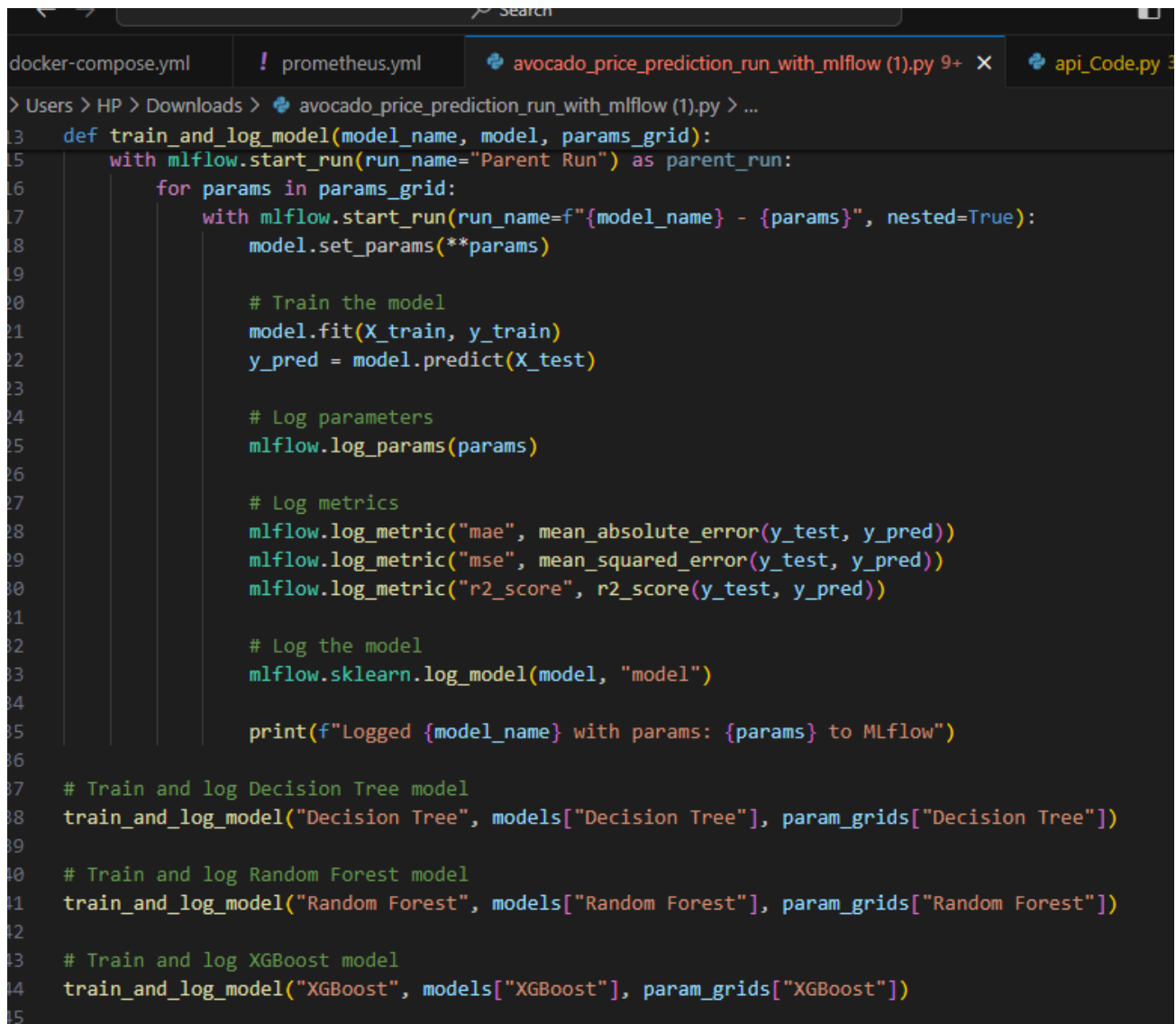
Flexibility: Can be adapted for different datasets and
Preprocessing requirements.

MLflow

Run **mlflow server**

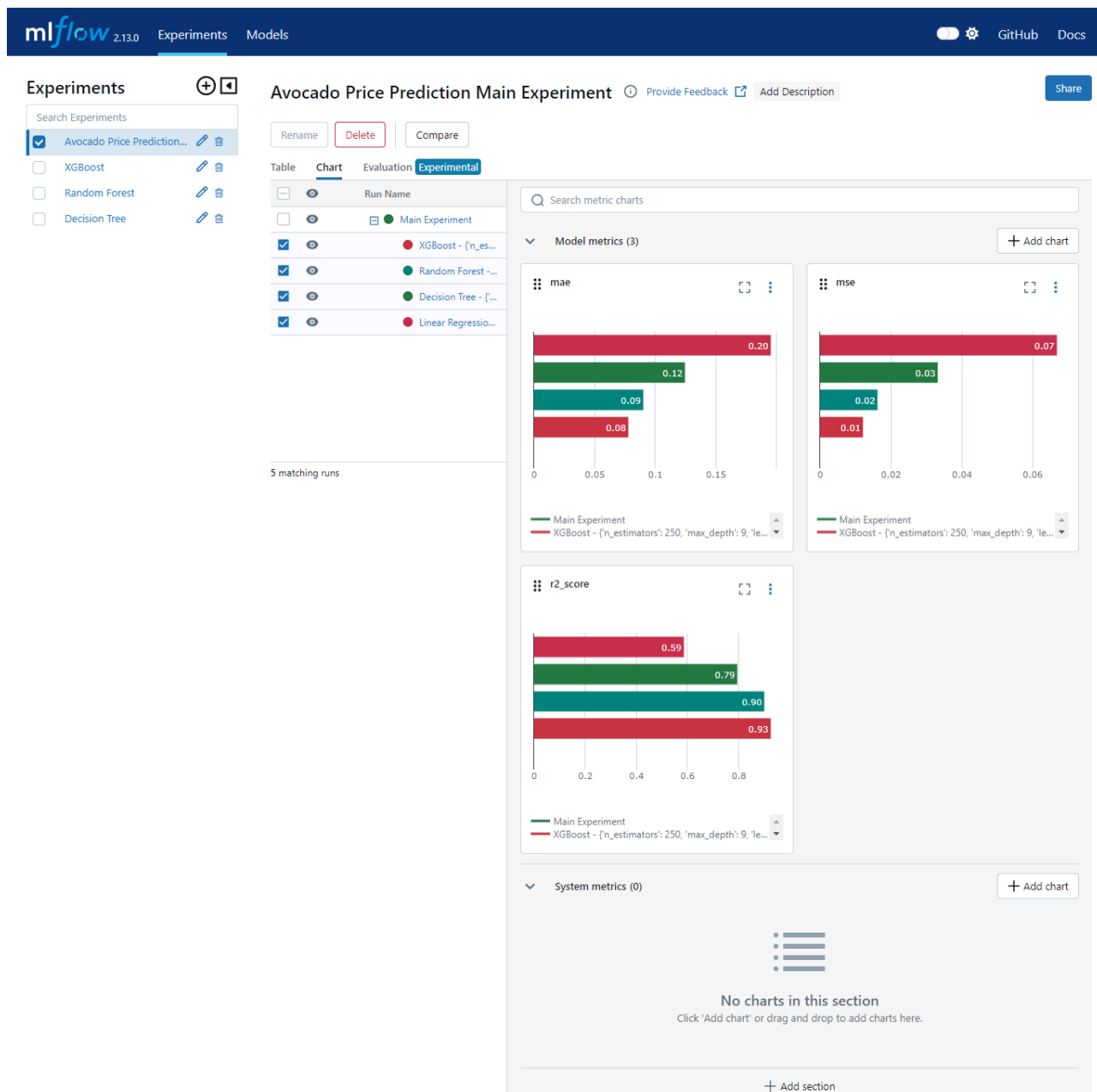
In the browser at <http://localhost:5000/>, the experiments conducted and metrics tracked can be visualized and recorded.

Code:

A screenshot of a code editor with a dark theme. The editor has several tabs at the top: 'docker-compose.yml', 'prometheus.yml', 'avocado_price_prediction_run_with_mlflow (1).py 9+', and 'api_Code.py 3'. The active tab is 'avocado_price_prediction_run_with_mlflow (1).py 9+'. The code is a Python script with line numbers 13 to 15 on the left. It defines a function 'train_and_log_model' that takes 'model_name', 'model', and 'params_grid' as arguments. The function uses 'mlflow.start_run' to create a parent run and then iterates over 'params_grid' to create nested runs. Inside each nested run, it sets parameters, trains the model, predicts on test data, and logs parameters and metrics (mae, mse, r2_score). It also logs the model using 'mlflow.sklearn.log_model'. Finally, it prints a message indicating the model and parameters are logged to MLflow. The script then calls 'train_and_log_model' for 'Decision Tree', 'Random Forest', and 'XGBoost' models using specific parameter grids.

```
13 def train_and_log_model(model_name, model, params_grid):
14     with mlflow.start_run(run_name="Parent Run") as parent_run:
15         for params in params_grid:
16             with mlflow.start_run(run_name=f"{model_name} - {params}", nested=True):
17                 model.set_params(**params)
18
19                 # Train the model
20                 model.fit(X_train, y_train)
21                 y_pred = model.predict(X_test)
22
23                 # Log parameters
24                 mlflow.log_params(params)
25
26                 # Log metrics
27                 mlflow.log_metric("mae", mean_absolute_error(y_test, y_pred))
28                 mlflow.log_metric("mse", mean_squared_error(y_test, y_pred))
29                 mlflow.log_metric("r2_score", r2_score(y_test, y_pred))
30
31                 # Log the model
32                 mlflow.sklearn.log_model(model, "model")
33
34                 print(f"Logged {model_name} with params: {params} to MLflow")
35
36
37 # Train and log Decision Tree model
38 train_and_log_model("Decision Tree", models["Decision Tree"], param_grids["Decision Tree"])
39
40 # Train and log Random Forest model
41 train_and_log_model("Random Forest", models["Random Forest"], param_grids["Random Forest"])
42
43 # Train and log XGBoost model
44 train_and_log_model("XGBoost", models["XGBoost"], param_grids["XGBoost"])
45
```

5 subruns for each model was done and the best models were created into the main experiment.



These experiments compare different hyperparameters of the models and determine the best model based on MSE, MAE, and R2 score metrics. A Main Experiment was then conducted to build and compare the best-performing models.

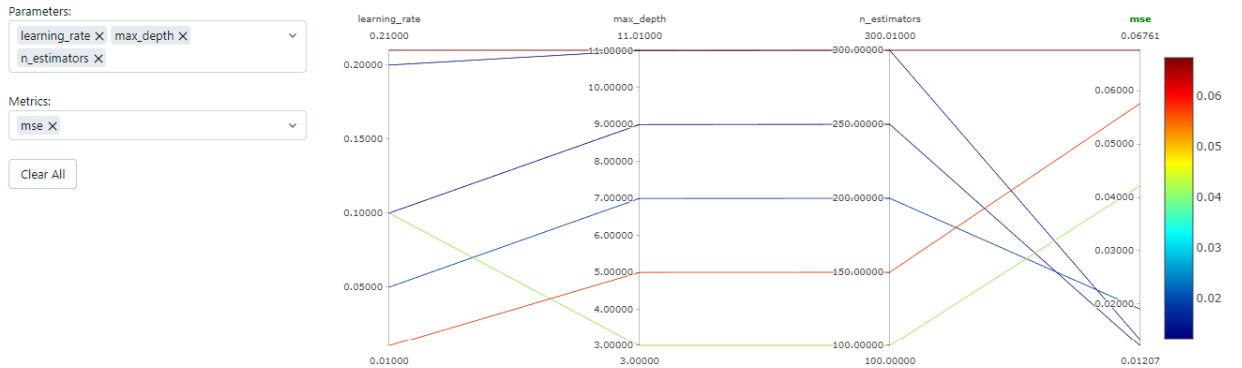
XGBoost-sub runs

XGBoost >

Comparing 6 Runs from 1 Experiment

Visualizations

Parallel Coordinates Plot Scatter Plot Box Plot Contour Plot



Run details

Run ID:	afcfb345f0a4c28bc9fbc03...	ff0588bff2804dc1be1773b...	53e73799ea3144ea83be6f...	f88a0bc5a6954d8d8a422c...	f2af5c1984a6446d800afe2...	8d39fc0c29da44539
Run Name:	Parent Run	XGBoost - {'n_estimators': 300, 'max_depth': 11, 'learning_rate': 0.2, 'subsample': 0.6}	XGBoost - {'n_estimators': 250, 'max_depth': 9, 'learning_rate': 0.1, 'subsample': 0.7}	XGBoost - {'n_estimators': 200, 'max_depth': 7, 'learning_rate': 0.05, 'subsample': 1.0}	XGBoost - {'n_estimators': 150, 'max_depth': 5, 'learning_rate': 0.01, 'subsample': 0.9}	XGBoost - {'n_estimators': 100, 'max_depth': 3, 'learning_rate': 0.1, 'subsample': 0.8}
Start Time:	2024-05-20 14:04:59	2024-05-20 14:05:18	2024-05-20 14:05:12	2024-05-20 14:05:08	2024-05-20 14:05:03	2024-05-20 14:04:59
End Time:	2024-05-20 14:05:27	2024-05-20 14:05:27	2024-05-20 14:05:18	2024-05-20 14:05:12	2024-05-20 14:05:08	2024-05-20 14:05:03
Duration:	28.0s	9.4s	6.0s	3.8s	5.5s	3.2s

Parameters

Show diff only

learning_rate	0.2	0.1	0.05	0.01	0.1
max_depth	11	9	7	5	3
n_estimators	300	250	200	150	100
subsample	0.6	0.7	1.0	0.9	0.8

Metrics

Show diff only

mae	0.081	0.078	0.102	0.185	0.157
mse	0.013	0.012	0.019	0.058	0.042
r2_score	0.918	0.925	0.882	0.643	0.739

Tags

Show diff only

No tags to display.

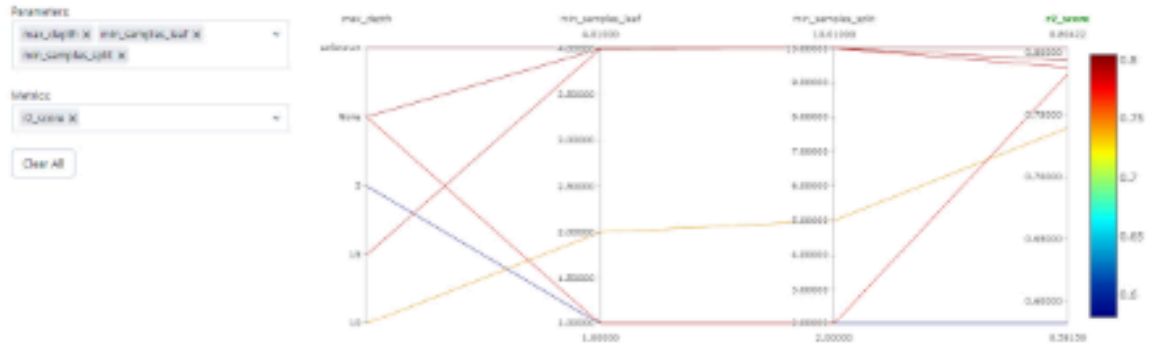
Decision Tree-sub runs

Decision Tree >

Comparing 6 Runs from 1 Experiment

Visualizations

Parallel Coordinates Plot Scatter Plot Box Plot Contour Plot



Run details

Run ID:	ab017b179b54a549f5b0c...	ab017b179b54a549f5b0c...	2a0b8d79e1010b18b780d...	c9a1b4a27f5b0a5b0f07e7...	8a4b5d89919a6a030a10b...	6a7d17b4080107349
Run Name:	Forest Run	Decision Tree - (max_depth: None, min_sample_split: 10, min_sample_leaf: 4)	Decision Tree - (max_depth: None, min_sample_split: 2, min_sample_leaf: 1)	Decision Tree - (max_depth: 15, min_sample_split: 10, min_sample_leaf: 4)	Decision Tree - (max_depth: 10, min_sample_split: 5, min_sample_leaf: 2)	Decision Tree - (max_depth: 3, min_sample_split: 10, min_sample_leaf: 1)
Start Time:	2024-05-20 14:02:48	2024-05-20 14:02:51	2024-05-20 14:02:48	2024-05-20 14:02:54	2024-05-20 14:02:51	2024-05-20 14:02:48
End Time:	2024-05-20 14:03:04	2024-05-20 14:03:04	2024-05-20 14:03:01	2024-05-20 14:03:08	2024-05-20 14:03:04	2024-05-20 14:03:01
Duration:	15.5s	2.4s	3.6s	3.7s	2.5s	3.0s

Parameters

Show diff only

max_depth	None	None	15	10	3
min_sample_leaf	4	1	4	2	1
min_sample_split	10	2	10	5	2

Metrics

Show diff only

r2_score	0.124	0.123	0.127	0.149	0.302
r2_score	0.093	0.095	0.094	0.042	0.098
r2_score	0.794	0.783	0.788	0.74	0.582

Deployment

FastAPI: FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python-type hints. It is designed to be easy to use and highly performant, making it ideal for creating APIs. In this project, FastAPI was used to create an API that processes image uploads and predicts digits using a pre-trained machine-learning model. The API endpoints include:

- **GET /:** A simple endpoint to check if the server is running.
- **POST /predict-task-2/:** An endpoint to upload an image, process it, and return the predicted digit.
- **GET /metrics:** An endpoint to expose Prometheus metrics.

Prometheus: Prometheus is an open-source monitoring and alerting toolkit for reliability and scalability. It collects and stores metrics as time series data. Prometheus was integrated with FastAPI to track API usage and performance metrics in this project.

Specifically:

- **Counters:** Track the number of API requests from different client IP addresses.
- **Gauges:** Monitor the processing time of the API concerning the input text length, measuring the effective processing time in microseconds per character.

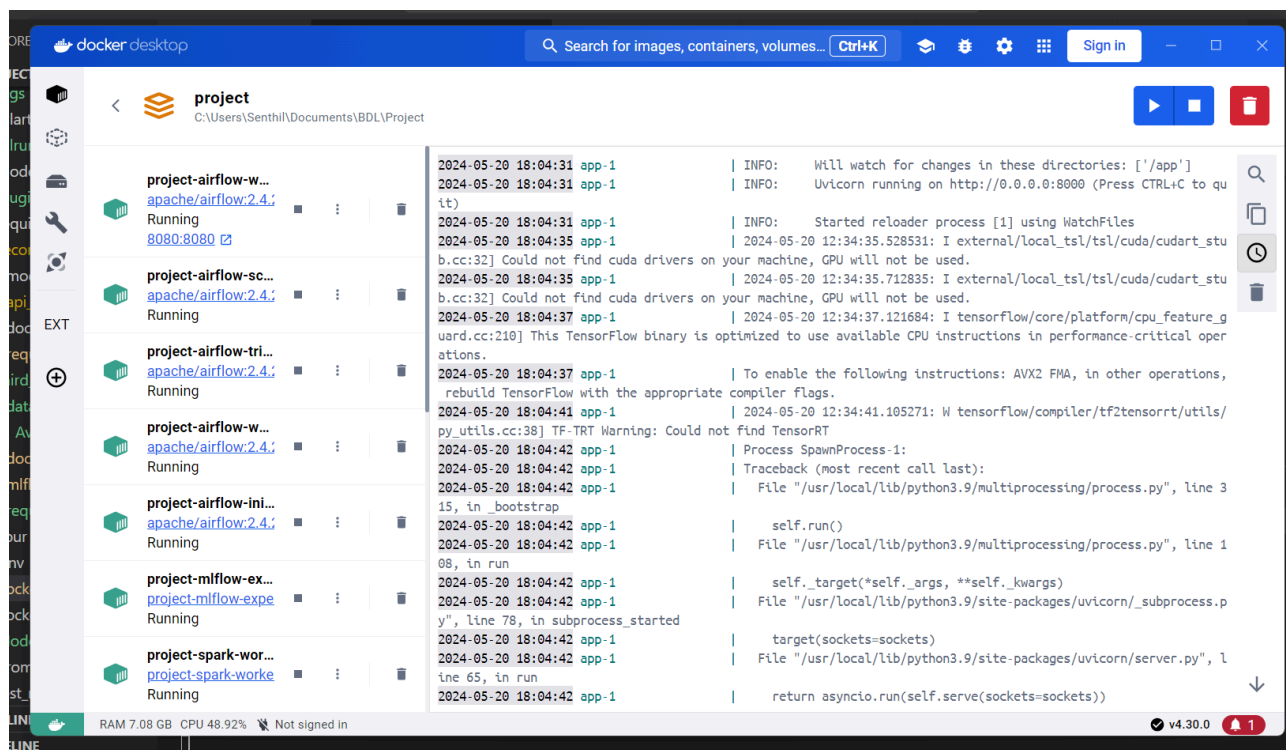
Grafana: Grafana is an open-source platform for monitoring and observability. It allows you to query, visualise, alert, and understand your metrics no matter where they are stored. In this project, Grafana was used to visualise the metrics collected by Prometheus. Users can monitor API performance and usage in real-time by creating dashboards and identifying trends and potential issues.

Docker: Docker is a platform that allows developers to automate the deployment of applications inside lightweight, portable containers. Docker ensures that applications run consistently across different environments. In this project, Docker containerised the FastAPI application and Prometheus and Grafana, making it

easier to deploy and manage these services together. The Docker setup typically involves:

- **Dockerfile:** Defines the FastAPI application's environment and dependencies.
- **docker-compose.yml:** Manages multi-container Docker applications, setting up FastAPI, Prometheus, and Grafana to work seamlessly.

Docker:



Fast API:

default

POST /predict Predict And Evaluate

Endpoint to predict values from an uploaded CSV file and calculate RMSE.

Parameters Cancel Reset

No parameters

Request body required multipart/form-data

file required Choose File Avocado_test.csv
string(\$binary)

Curl

```
curl -X 'POST' \
  'http://localhost:8000/predict' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@avocado_test.csv;type=text/csv'
```

Request URL

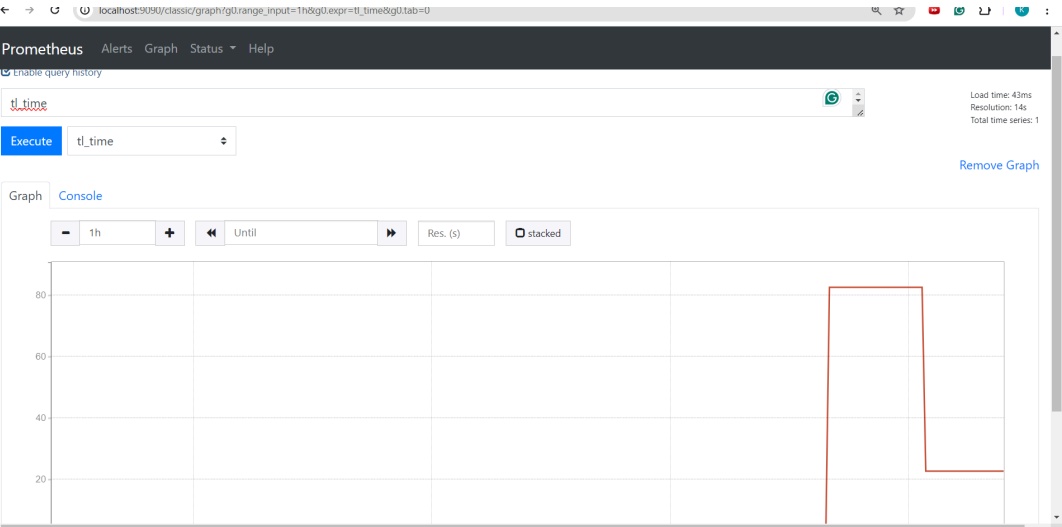
http://localhost:8000/predict

Server response

Code	Details
200	<div>Response body</div> <pre>{ "mae": 0.36857783491958046, "mse": 0.2186617199422799 }</pre> <div>Download</div> <div>Response headers</div> <pre>content-length: 52 content-type: application/json date: Sat, 18 May 2024 14:32:21 GMT server: uvicorn</pre>

Responses

Prometheus:



localhost:9090/classic/targets

Prometheus Alerts Graph Status Help

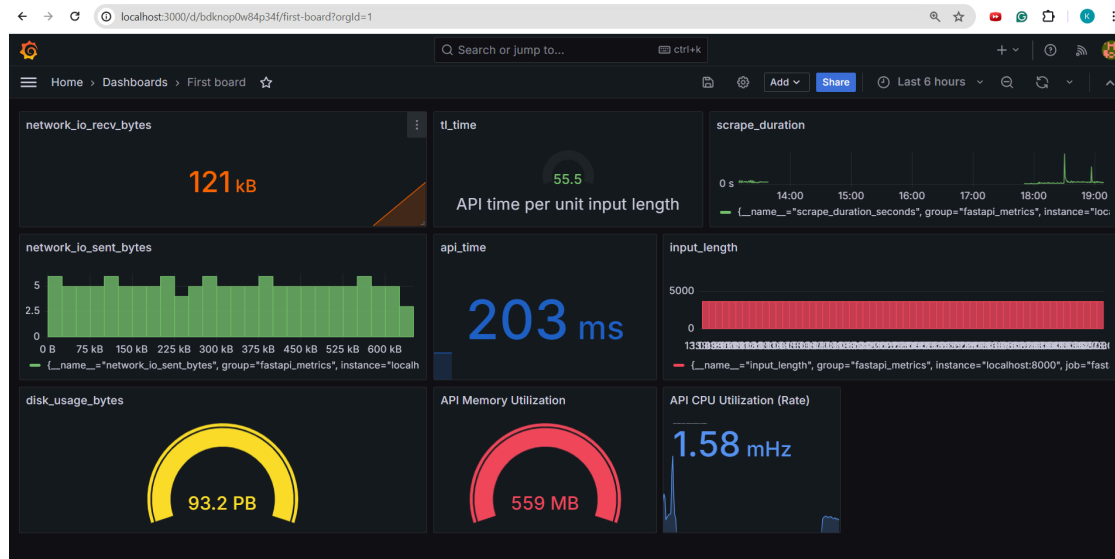
Targets

All Unhealthy Collapse All

fastapi (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:8000/metrics	UP	group="fastapi_metrics" instance="localhost:8000" job="fastapi"	2.175s ago	4.115ms	

Grafana:



Integration Details

1. FastAPI Integration:

- Created an API with endpoints for predicting digits (`/predict-task-2/`) and exposing metrics (`/metrics`).

2. Prometheus Integration:

- Integrated Prometheus metrics by adding Counters and Gauges in the FastAPI application.
- Exposed Prometheus metrics through the `/metrics` endpoint.

3. Grafana Integration:

- Configured Grafana to connect to Prometheus as a data source.
- Created dashboards in Grafana to visualize metrics such as API request counts and processing times.

4. Docker Integration:

- Used Docker to containerize the FastAPI application, ensuring consistent deployment.
- Created a `docker-compose.yml` file to set up FastAPI, Prometheus, and Grafana containers.

Endpoints

Docker file

```
second_docker > dockerfile > ...
1  # Switch to a Python base image for the Flask application
2  FROM python:3.9
3
4  # Set the working directory for the Flask application
5  WORKDIR /app
6
7  # Copy the current directory contents into the container at /app
8  COPY . /app
9
10 # Install Python dependencies for the Flask application
11 RUN pip install --no-cache-dir -r requirements.txt
12
13 # Expose port 8000 for Flask app
14 EXPOSE 8000
15
16 # Set environment variables
17 ENV PYTHONUNBUFFERED=1
18
19 # Specify the command to run the Flask application with Uvicorn
20 CMD ["uvicorn", "api_code:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]
21
```

Docker-compose file

```
services:
  app:
    build:
      volumes:
        - C:/Users/Senthil/Documents/BDL/Project/second_docker/model/xgboost_model.pkl:/app/model/xgboost_model.pkl
    ports:
      - "8000:8000"
    mem_limit: 512m
    cpu_count: 1

  prometheus:
    image: prom/prometheus
    ports:
      - "9090:9090"
    volumes:
      - "C:/Program Files/prometheus-2.52.0.windows-amd64/prometheus.yml:/etc/prometheus/prometheus.yml"
    command: --config.file=/etc/prometheus/prometheus.yml

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"
```

To run the program

- Docker-compose up --build

Contributions:

Version control: **Git** and **Git-LFS**

- **Keerthana (ED19B014):**

Apache Airflow Deployed APIs with **FastAPI**, tracked the API usage with **Prometheus** and visualisation with **Grafana**, and containerized the whole thing using **docker**, Monitoring Dashboard using **Grafana**.
Tracking the project using git.

- **Rinkle (NA19B056):**

PS and dataset selection, analysis, **Preprocessing Pipeline** with **Apache Spark** and modelling and validation experiments, tracking these using **MLFlow**. Tracking the project using git.

- **Aravind (CE19B099):**

Model Selection and data analysis.