HOUSE RENT PREDICTION

MR.JEDNIPAT	KEMAWAT	64070501011
MR.CHAYAROB	CHANTRAPIWAT	64070501015
MR.TECHATHAT	SAKULSAK	64070501064
MS.SARUNYARAT	WONGSASON	64070501086
MR.WORAPOL	KHUNAEKANAN	64070501097

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR

CPE393 SPECIAL TOPIC III : MACHINE LEARNING OPERATIONS

FACULTY OF ENGINEERING

KING MONGKUT'S UNIVERSITY OF TECHNOLOGY THONBURI

2024

Table of Contents

Chapter 1: Project Overview

This chapter introduces the project's background, objectives, and the importance of implementing an MLOps workflow. It sets the context for the problem addressed and outlines the project goals, team collaboration, and overall impact.

1.1 Objectives

- Implement a full MLOps workflow: Cover the entire machine learning lifecycle, starting from data preparation, moving through model training and evaluation, and culminating in model deployment.
- Use experiment tracking tools: Employ experiment tracking platforms (such as MLflow) to monitor, compare, and improve model performance, ensuring all experiments are logged and reproducible.
- Collaborate efficiently as a team: Utilize collaborative tools, clear responsibility assignment, and best practices in version control to manage and deliver the machine learning project effectively as a team.

1.2 Problem Definition

The goal of this project is to predict rental prices for residential properties in six major cities of India: Mumbai, Chennai, Bangalore, Hyderabad, Delhi, and Kolkata. Accurate rent predictions are vital for tenants, landlords, real estate agents, researchers, and policymakers. Tenants can make informed decisions and avoid overpaying, while landlords and agents can set competitive rental rates based on market trends and property features. Researchers and policymakers gain insights into housing markets to inform urban development and policy. By using a dataset of property features such as location, size, and number of rooms, the project aims to build a robust machine learning model that accurately estimates rental prices, supporting better decision-making across the housing sector.

1.3 Team Organization

Student ID	Name	Responsibility
64070501011	Jednipat Kemawat	Deployment
64070501015	Chayarob Chantrapiwat	Deployment
64070501064	Techathat Sakulsak	Model Training
64070501086	Sarunyarat Wongsason	Preprocessing
64070501097	Worapol Khunaekanan	Model Training

1.4 Tools Used

To build the predictive model and facilitate the MLOps workflow, the following tools and technologies were utilized:

- **Google Colab:** A cloud-based Jupyter notebook environment that allowed for easy collaboration, access to GPUs for model training, and seamless integration with Python libraries such as Pandas, Scikit-learn, and TensorFlow.
- **Airflow:** A platform for orchestrating data pipelines, ensuring that the data preparation and model training tasks are automated, reproducible, and scalable.
- **MLflow:** Used for experiment tracking and model management, MLflow helps in logging hyperparameters, metrics, and artifacts, making it easy to compare different models and ensure reproducibility.
- **Docker:** Containerization was utilized to create consistent and isolated environments for model development and deployment, ensuring that the application behaves the same way across different systems.
- **Flask:** A lightweight web framework used to develop a REST API for deploying the predictive model, allowing users to make predictions about rental prices via a web interface.
- **HTML, CSS, JS, Python:** The front-end (HTML, CSS, JavaScript) and back-end (Python) were used to build a user-friendly web interface that interacts with the Flask API, allowing users to input property details and get rental price predictions.

1.4 Project Scope

The scope of this project covers the entire MLOps lifecycle:

1. **Data Collection:** Gathering relevant datasets, including property details (e.g., location, size, amenities) and market trends.
2. **Data Preprocessing:** Cleaning and transforming the data to ensure it is suitable for model training.
3. **Model Training & Evaluation:** Applying machine learning algorithms to train the model, followed by rigorous evaluation using metrics such as MSE, MAE, and R^2.
4. **Experiment Tracking:** Using MLflow to track experiments, log metrics, and ensure reproducibility.
5. **Model Deployment:** Deploying the trained model as a production-ready API

1.5 Project Impact

- **For Renters**: The predictive model offers a tool for estimating fair rental prices, allowing renters to make more informed decisions and negotiate better terms.
- **For Landlords & Real Estate Agents**: The model provides data-driven insights to help landlords set competitive rental rates and agents offer properties that align with market expectations.
- **For Researchers & Policymakers**: The dataset and modeling framework provide valuable information for studying urban housing markets and formulating effective housing policies.
- **Technical Impact**: This project demonstrates the power of MLOps practices, incorporating tools such as MLflow, Airflow, Docker, and CI/CD pipelines to ensure scalability, reproducibility, and maintainability of machine learning systems.

Chapter 2: Data Pipeline and Feature Engineering

This chapter outlines the processes involved in collecting, processing, and preparing the data for model training. It covers the feature engineering process, which includes data cleaning, transformation, and selection, ensuring the dataset is of the highest quality for downstream machine learning tasks.

2.1 Data Collection & Labeling

India's housing market is diverse, ranging from luxury properties to modest rural homes. This diversity reflects the country's rapidly growing housing sector, driven by increasing income levels. Despite the expansion, access to adequate housing remains a challenge, with only 60.9% of the housing rights potential being fulfilled, according to the Human Rights Measurement Initiative. In this context, renting plays a crucial role in India's urban housing market by providing greater accommodation access and contributing to the sharing economy.

2.1.1 Source

The dataset used in this project, the House Rent Prediction Dataset, is publicly available on Kaggle. It was originally collected through web scraping from MagicBricks, a leading Indian real estate platform. The dataset comprises over 4,700 residential rental listings, which include a variety of properties such as houses, apartments, and flats. Each listing includes the following attributes:

- **BHK**: Number of Bedrooms, Hall, and Kitchen.
- **Rent**: Monthly rental price (target variable).
- **Size**: Area of the property in square feet.
- **Floor**: The floor on which the property is located and the total number of floors (e.g., "Ground out of 2", "3 out of 5").
- **Area Type**: Type of area measurement (e.g., Super Area, Carpet Area, or Built Area).
- **Area Locality**: The neighborhood or locality where the property is situated.
- **City**: The city in which the property is located.
- **Furnishing Status**: The level of furnishing (e.g., Furnished, Semi-Furnished, or Unfurnished).
- **Tenant Preferred**: The type of tenant preferred by the property owner (e.g., family, bachelor).
- **Bathroom**: The number of bathrooms in the property.
- **Point of Contact**: Contact details for inquiries related to the listing.

2.1.2 Annotate/labeled data

This is a supervised learning problem, where the target variable is Rent (the monthly rental price). The dataset is already labeled, as each entry includes the rental price associated with the property listing.

2.1.3 Data Ingestion

The dataset is provided in CSV format, where each row represents a unique rental listing and each column corresponds to one of the attributes mentioned above. This format facilitates easy ingestion into the data pipeline for further processing and analysis.

2.2 Training Dataset Curation

2.2.1 Data Cleaning

Data cleaning is the foundational step to ensure the quality of the dataset before model training. In the provided data processing pipeline, cleaning involves:

- **Outlier Removal:** Outliers can significantly skew model performance. Using the Interquartile Range (IQR) method, the script identifies and removes values that fall outside the acceptable range, particularly in important numerical features like `Rent` and `Size`. This step ensures that the model is not overly influenced by extreme values that don't represent typical data points.
- **Missing and Irregular Value Handling:** Handling missing or irregular values is essential to prevent errors during model training. In the case of columns like `Floor`, non-numeric values such as "Ground" and "Basement" are converted to numeric codes. Missing values are filled based on statistical methods (e.g., mean or median) or, in some cases, replaced with a placeholder indicating absence, ensuring all data can be processed without breaking the pipeline.
- **Dropping Unnecessary Columns:** Irrelevant columns, such as `Posted On` and `Area Locality`, are removed from the dataset. This reduces the dimensionality of the dataset and focuses the model only on the most relevant features that contribute directly to prediction.

2.2.2 Dataset Splitting

Splitting the dataset into training and testing sets is crucial to validate model performance and ensure generalizability:

Train/Test Split: The pipeline uses an 80/20 split (with `train_test_split` from scikit-learn), which means 80% of the data is used for training and 20% for testing. This ensures that the model is trained on a diverse set of examples and evaluated on unseen data, minimizing the risk of overfitting and giving a more realistic estimate of its real-world performance.

2.2.3 Versioning

Proper versioning practices ensure that models, transformations, and data are reproducible across different runs and experiments:

- **Encoder and Scaler Versioning:** Both the `StandardScaler` and `OneHotEncoder` are fitted on the training data and saved as `.pkl` files. This versioning approach ensures that the

transformations applied during training are exactly the same during inference, enabling consistent predictions. These files are stored in a dedicated `models` directory for easy access and management.

2.3 Feature Engineering

2.3.1 Feature Selection

Feature engineering is the process of transforming raw data into meaningful features that improve model performance:

- **Categorical Feature Encoding:** Categorical variables, such as `Area Type` and `City`, are transformed using one-hot encoding. This converts the data into a binary format (0 or 1), making it suitable for machine learning models that require numerical input. Each category is represented as a separate binary feature, preserving the information while enabling effective use of algorithms.

- **Numerical Feature Scaling:** Features like `BHK`, `Size`, `Bathroom`, `CurrentFloor`, and `TotalFloors` are scaled using `StandardScaler` to ensure they all contribute equally to the model. By scaling to zero mean and unit variance, the model is not biased by features with larger numeric ranges, and the learning process is more stable and efficient.

- **Floor Feature Extraction:** The `Floor` feature is particularly complex because it includes both the current floor and the total number of floors in a building (e.g., "5th of 10"). To make this data usable, the script splits this information into two numerical columns: one representing the current floor and the other representing the total number of floors. This extraction enables the model to better understand the relationship between these two features and their potential impact on rental prices.

2.3.2 Transformation

Transformation ensures that the data is properly prepared for the model:

- **Feature Concatenation:** After encoding categorical features and scaling numerical ones, the processed features are concatenated into a single matrix. This matrix serves as the final input for the machine learning model. This step combines both the high-dimensional categorical data and normalized numerical data into a unified structure, ready for model training.

- **Consistent Transformation for Inference:** Consistency is key when moving from training to inference. The same encoders and scalers that were used to process the training data are saved and reloaded during inference. This guarantees that the data passed to the model for prediction is transformed in exactly the same way as the training data, preserving model accuracy.

2.4 Data Pipeline

Step	Description
1. Data Source	Public dataset from Kaggle, collected via web scraping from MagicBricks.
2. Data Ingestion	Load and inspect the raw CSV file; identify the target variable (Rent).
3. Data Cleaning	Remove outliers, handle missing/improper values, drop unnecessary columns.
4. Train/Test Splitting	Divide the dataset into 80% training and 20% testing sets.
5. Feature Engineering	Encode categorical variables, scale numerical features, extract meaningful derived features.
6. Feature Transformation	Combine all processed features into a single input matrix for modeling.
7. Versioning & Saving	Serialize preprocessing artifacts (OneHotEncoder,StandardScaler) for future use.
8. Consistent Inference	Reuse saved preprocessing tools during deployment to maintain prediction accuracy.

Chapter 3: Model Training Results

This chapter presents a comprehensive overview of the model training and evaluation process, emphasizing experiment tracking and model comparison. It also justifies the selection of the final model based on performance metrics and generalization capabilities.

3.1 Model Training & Experiment Tracking

3.1.1 Models Compared

1. **Random Forest**

   Overview: Random Forest is an ensemble-based learning method that constructs multiple decision trees and aggregates their outputs to improve accuracy. It is particularly effective in managing overfitting by averaging multiple trees, thereby providing more robust predictions.

   Use Case: Well-suited for capturing complex non-linear relationships and handling high-dimensional data without the need for feature scaling.

2. **Ridge Regression:**

   Overview: Ridge Regression is a type of linear regression with L2 regularization. It helps mitigate multicollinearity and overfitting, making it effective in high-dimensional datasets or when there is a need for controlling the influence of outliers.

   Use Case: Ideal for linear relationships and when model interpretability (coefficients) is important.

3. **Neural Network**

   Overview: A feedforward neural network that uses multiple layers of neurons to capture complex non-linear relationships between the input features and the target. Neural networks are flexible but require careful tuning and large datasets to perform optimally.

   Use Case: Effective for highly complex, non-linear problems, but can struggle with smaller datasets or poor architecture choices.

3.1.2 Experiment Tracking

To ensure reproducibility and track all experiment parameters, **MLflow** was used for experiment tracking. MLflow helps in logging and organizing the following aspects:

- **Metrics**: Logs key performance metrics like MSE, MAE, and R^2 for each model iteration, making it easy to compare results.
- **Artifacts**: Stores models, trained data, and any other relevant artifacts generated during the experiment.

- **Reproducibility**: All training and evaluation scripts were tracked using **Git**, which guarantees the experiments can be recreated at any point in time.

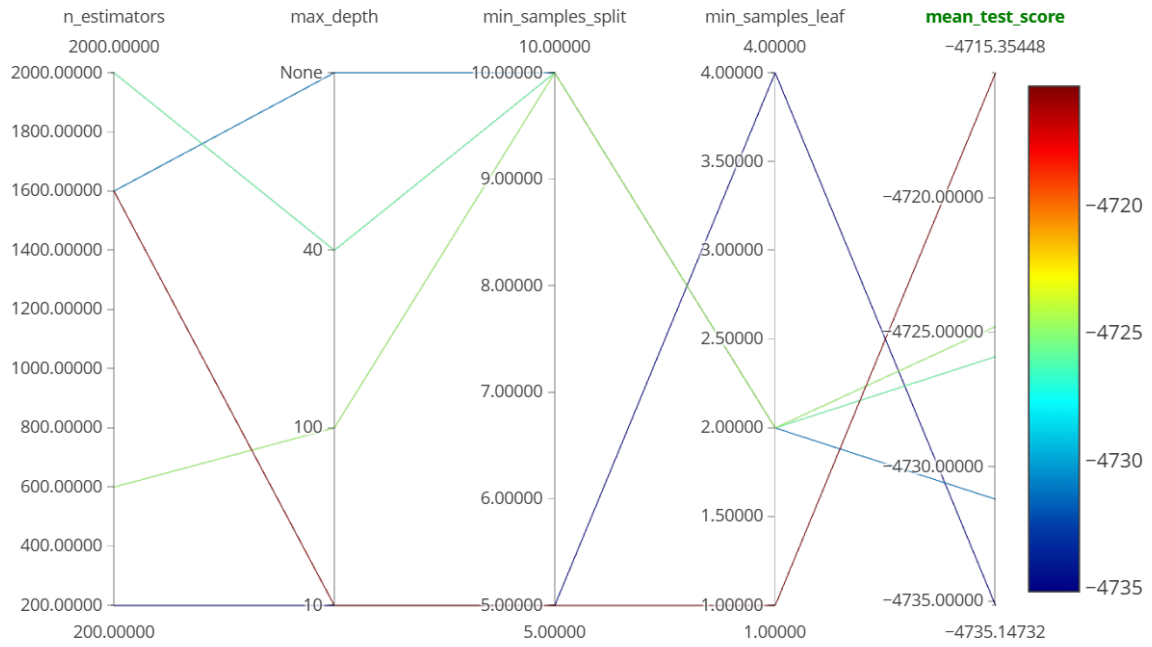
3.1.3 Hyperparameter Tuning

1. **Random Forest**

Utilize sklearn's RandomizedSearchCV to search through a hyperparameter grid and log the 5 best models to MLflow. The hyperparameters used is as follows:

- n_estimators: 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000
- max_depth: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, None
- min_samples_split: 2, 5, 10
- min_samples_leaf: 1, 2, 4

The following figure shows the parallel coordinates plot of the best 5 models, monitoring the metric of neg_mean_absolute_error.



The best hyperparameters are as follows:

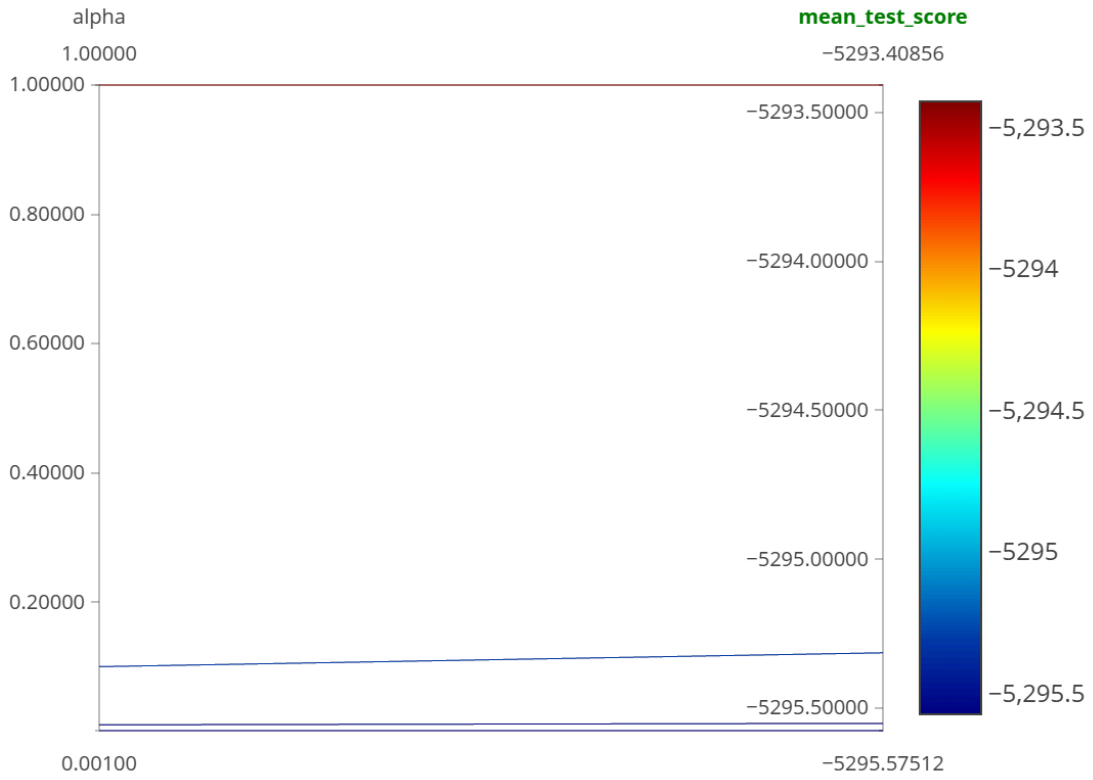
max_depth	None
min_samples_leaf	2
min_samples_split	10
n_estimators	1600

2. Ridge Regression

Utilize sklearn's GridSearchCV to search through a hyperparameter grid and log all 4 models to MLflow. The hyperparameters used is as follows:

- alpha: 0.001, 0.01, 0.1, 1.0

The following figure shows the parallel coordinates plot of all models, monitoring the metric of neg_mean_absolute_error.



The best hyperparameters are as follows:

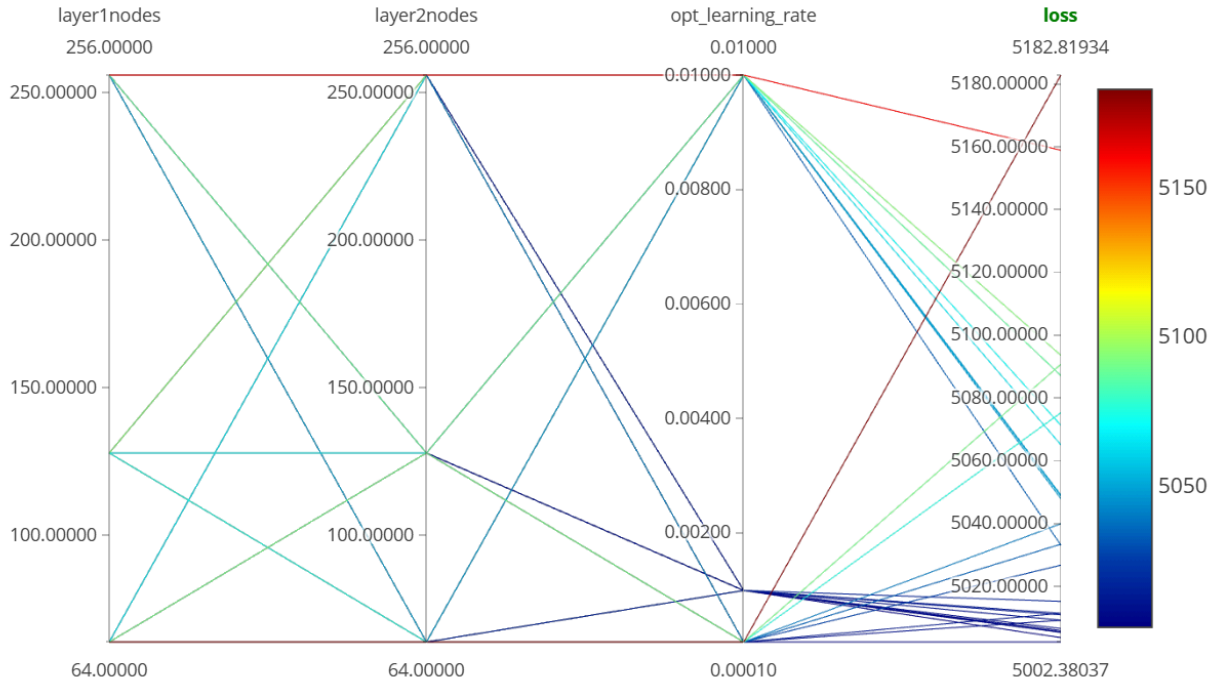
alpha	1.0

3. Neural Network

Utilize TensorFlow to search through a hyperparameter grid and log all 27 models of 2-layer dense neural networks to MLflow. The hyperparameters used is as follows:

- layer1_node: 64, 128, 256
- layer2_node: 64, 128, 256
- learning_rate: 0.0001, 0.001, 0.01

The following figure shows the parallel coordinates plot of all 27 models, monitoring the metric of loss.



The best hyperparameters are as follows:

layer1nodes	256
layer2nodes	256
opt_learning_rate	0.001

3.1.4 Evaluation Process

Each model underwent the following process:

1. **Training**: Models were trained using the training dataset.
2. **Evaluation Metrics**: Performance was assessed using Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R^2) for both the training and test datasets.

3.1.5 Evaluation Results

Training Set Performance

Model	Mean Squared Error (MSE)	Mean Absolute Error (MAE)	R-squared (R^2)
Random Forest	**19469060.27**	**3247.56**	**0.89**
Ridge	54334118.84	5256.04	0.69
Neural Network	57514526.23	5102.12	0.67

Random Forest shows the best performance, with MAE value of 3247.56 and R^2 value of 0.89. This indicates that both models fit the training data well and capture the underlying patterns effectively.

Neural Network and Ridge exhibit lower performance, with MAE values of 5102.12 and 5256.04, and R^2 values of 0.67 and 0.69 respectively. This could indicate that these models are inappropriate for the data .

Test Set Performance

Model	Mean Squared Error (MSE)	Mean Absolute Error (MAE)	Coefficient of determination (R-squared)
Random Forest	**41514612.62**	**4452.25**	**0.77**
Ridge	51513004.86	5244.22	0.71
Neural Network	53446249.89	5070.60	0.70

Similarly to the training set, Random Forest shows the best performance, with MAE value of 4452.25 and R^2 value of 0.77. Neural Network performs the second-best with MAE value of 5070.60, followed by Ridge with MAE value of 5244.22.

3.1.5 Best Model Selection

Based on the evaluation across both training and test sets, Random Forest significantly outperforms Ridge Regression and Neural Network models, with the lowest MAE and highest R^2 value. There could be several factors that make it the better choice for this particular task, chiefly that the problem statement requires predicting continuous variables from limited features, with the majority being categorical. This is similar to how Random Forest naturally works, while Neural Networks would perform worse due to the limited information, and Ridge Regression would perform worse due to its simplistic nature being unable to capture the dataset's trend.

Therefore, considering the overall performance and the nature of all three models, Random Forest was determined to be the most suitable model for this analysis and is selected as the best model.

Chapter 4 Deployment setup

This chapter provides an overview of how the best-performing machine learning model was deployed as a production-ready service. The deployment setup leverages REST APIs, containerization, and orchestration tools to create a scalable, serverless system. It also covers key practices such as CI/CD (if applicable) and monitoring for maintaining the service in a production environment.

4.1 API Service

4.1.1 GET /health

Description:

The GET /health endpoint is used to check the health and availability of the deployed service. It is commonly used for monitoring purposes to verify that the API is operational and the backend service is running without issues.

Response:

- Status Code: 200 OK
- Content-Type: application/json
- Request Body Example:

```
{
    "status": "healthy"
}
```

4.1.2 POST /predict

Description:

The POST /predict endpoint accepts a set of features related to a rental property and returns the predicted rental price. The input features include parameters such as the property's location, size, furnishing status, and other relevant information.

Request:

- Method: POST
- Content-Type: application/json
- Request Body Example:

```
{
    "BHK": 1,
    "Size": 1000,
    "CurrentFloor": -1,
    "TotalFloors": 2,
    "Area Type": "Super Area",
    "City": "Mumbai",
    "Furnishing Status": "Semi-Furnished",
    "Tenant Preferred": "Family",
    "Bathroom": 1,
    "Point of Contact": "Contact Agent"
}
```

- Parameter Describe
    - BHK: Number of Bedrooms, Hall, and Kitchen (positive integer)
    - Size: Area size in square feet (positive integer)
    - CurrentFloor: Number of floor available (0 for ground, -1 for underground, and other positive integer)
    - TotalFloors: Total number of floors in the building (positive integer)
    - Area Type: Type of area (Super Area, Carpet Area, Built Area)
    - City: The city located (Mumbai, Bangalore, Delhi, Hyderabad, Chennai, Kolkata)
    - Furnishing Status: Furnishing status (Furnished, Semi-Furnished, Unfurnished)
    - Tenant Preferred: Preferred tenants: (Bachelors, Bachelors/Family, Family)
    - Bathroom: Number of bathroom (positive value)
    - Point of Contact: Contact type (Contact Owner, Contact Agent, Contact Builder)

Response:

- Status Code: 200 OK
- Content-Type: application/json
- Body:

```
{
    "prediction": {
        "confidence_interval": {
            "lower": 10024.574844449988,
            "upper": 66465.42515555001
        },
        "prediction": 38245.0
    },
```

```
    "success": true
}
```

4.2 Containerization and Orchestration

The deployment of the model is containerized and orchestrated using Docker and AWS Lambda. The use of containerization ensures that the environment is consistent, making it easier to deploy and scale in the cloud.

4.2.1 Local Deployment

To use the application, you can either access it via the deployed cloud endpoint at https://wt9vo8xuke.execute-api.ap-southeast-1.amazonaws.com/dev/ , which is described in more detail in the following section, or run it locally on your machine. To deploy locally, please follow these steps:

- Clone the repository: `git clone https://github.com/WorapolKhu/CPE393-group_name`
- Build the Docker image: `docker build -t rent-prediction .`
- Run the Docker container: `docker run -p 5000:5000 rent-prediction`
- Access the application at: `http://localhost:5000`

4.2.2 Docker

The entire model, preprocessing logic, and Lambda handler are packaged into a Docker container to ensure portability and consistency across different environments. The container is built using the official AWS Lambda Python 3.13 base image.

The Dockerfile is divided into two stages:

- **Builder stage**: Installs dependencies from `requirements.txt`.
- **Final Stage:** Copies the installed dependencies and application code into a lightweight container.

Dockerfile: (for AWS deployment)

```dockerfile
# --- Builder Stage ---
FROM public.ecr.aws/lambda/python:3.13 AS builder

ENV TMPDIR=/tmp

# Copy requirements file
COPY requirements.txt .

# Install all dependencies
RUN pip install --upgrade pip --no-cache-dir && \
    pip install -r requirements.txt --no-cache-dir && \
    rm -rf /tmp/* /var/tmp/*

# --- Final Stage ---
FROM public.ecr.aws/lambda/python:3.13

# Copy only installed packages from builder
COPY --from=builder /var/lang/lib/python3.13/site-packages /var/lang/lib/python3.13/site-packages

# Copy app source code
COPY lambda_function.py .
COPY data_processing.py .
COPY templates/ ./templates

# Set Lambda handler
CMD ["lambda_function.lambda_handler"]
```

4.2.3 AWS Cloud Deployment

After the Docker image is pushed to Amazon Elastic Container Registry (ECR), the model is deployed using a serverless architecture via AWS Lambda, and it is made publicly accessible through Amazon API Gateway. Additionally, a Flask web interface can be hosted to interact with the model through the `/predict` API route, using API Gateway as a proxy.

Key Services Used:

- **Amazon ECR (Elastic Container Registry):** Hosts the Docker image, which is used by AWS Lambda to execute the model.
- **AWS Lambda (Container Image)**: Runs the model as a serverless function using the container image from Amazon ECR.
- **Amazon API Gateway**: Provides **RESTful** endpoints to interact with the Lambda function. It acts as a reverse proxy for routing HTTP requests to Lambda.
- **Amazon CloudWatch**: Collects logs and metrics for monitoring, debugging, and performance analysis.

4.2.4 Deployment Steps

This section details the step-by-step process for deploying the model to AWS using Docker,

Lambda, and API Gateway. The deployment ensures that the system is highly scalable, serverless, and easily maintained.

1. **Push Docker Image to ECR**

   Build the Docker image as described in the Docker section and push it to Amazon ECR for storage.

2. **Create AWS Lambda Function (Using Container Image)**
   - Choose `Container image` as the deployment method in AWS Lambda.
   - Set the image URI from Amazon ECR for the Lambda function to use the container image.
   - Set the Lambda function handler to `lambda_function.lambda_handler`.
3. **Create and Configure API Gateway**
   - Create a new REST API in API Gateway.
   - Set up proxy integration to forward any HTTP method and route (e.g., `/predict`) directly to Lambda.
   - Define routes:
     - `GET /health`: Used for health checks to ensure the system is running.
     - `POST /predict`: The main endpoint for predicting rental prices.
   - Enable proxy integration so that routing behaves like Flask-style routing, passing requests directly to Lambda.
4. **Connect API Gateway to Lambda**
   - Use **Lambda Integration** to connect API Gateway to the Lambda function.
   - Set the deployed container-based Lambda as the integration target.
5. **Enable Logging with CloudWatch**
   - Enable **CloudWatch logs** and **metrics** to monitor the system.
   - This includes logging **request traffic**, **tracking error messages**, and **monitoring performance issues** (e.g., execution time, function timeouts).

Chapter 5: Challenges and Potential Solutions

This chapter analyzes the main challenges encountered during the project and discusses strategies and solutions to address them. It reflects on lessons learned and provides recommendations for future work.

5.1 Challenges and Potential Solutions

5.1.1 Data Quality and Inconsistencies

Challenge:
The dataset used in the project contained various data quality issues, including missing values and ambiguous categorical values. For instance, the "Floor" field included inconsistent labels that made standardization difficult. These inconsistencies negatively affected model performance by introducing noise and reducing the reliability of feature representations.

Solution:
To address these issues, a comprehensive data cleaning process was conducted. Missing values were handled using appropriate imputation techniques or removed when necessary. Categorical variables were standardized and encoded using consistent schemes like OneHotEncoding. Numerical fields were normalized using MinMaxScaler, and units were made uniform.

5.1.2 Limited Dataset Size

Challenge:

The dataset consisted of approximately 4,700 samples, which is relatively small for training machine learning models, especially deep learning algorithms that typically require larger datasets. With limited data, there is a higher risk of overfitting, reduced generalization, and poor model stability.

Solution:

To mitigate the effects of the limited dataset, advanced feature engineering was employed. This included outlier removal, categorical encoding, and numerical scaling to enhance data quality and boost model learning. Random Forest was chosen as the primary model due to its robustness and capability to perform well on small datasets. Its ensemble approach helped reduce overfitting and provided a balanced performance despite the dataset constraints.

5.1.3 Containerization and Cloud Integration

Deploying the trained model as a REST API using Docker and AWS Lambda required careful configuration to ensure compatibility and scalability. Issues such as dependency management, cold starts in Lambda, and API Gateway integration posed challenges during deployment.

Solution:

A multi-stage Dockerfile was designed to manage dependencies efficiently, and lightweight libraries were prioritized to reduce container size. The deployment process was streamlined using Amazon ECR and Lambda's container support, ensuring seamless integration with API Gateway.

5.1.4 Reproducibility Across Environments

Differences in development environments led to occasional inconsistencies in code execution, particularly during the transition from local development to cloud deployment.

Solution: Docker was used to create isolated and consistent environments across all stages of the project. Requirements.txt files and virtual environments were also maintained to ensure dependency consistency.

5.2 Lessons Learned

Throughout the course of this project, several key lessons were learned. Starting with simpler models like Random Forest proved effective, especially with limited data, offering strong performance without the complexity of deep learning approaches. Data quality emerged as a critical factor in model success, highlighting the importance of thorough preprocessing steps such as handling missing values, encoding categorical variables, and scaling features. Ensuring reproducibility through saving preprocessing artifacts and using Docker containers helped maintain consistency between training and inference stages. Additionally, structured project management, including clearly defined roles, Git-based version control, and regular communication, was vital for effective collaboration. These experiences underscored the importance of simplicity, data integrity, reproducibility, and teamwork in developing reliable machine learning systems.

5.3 Future Work

Future work can focus on several key areas to enhance the current system. Expanding the dataset, either through data augmentation or integration of external sources, could improve model performance and generalizability.

Appendix

References

- **Amazon ECR | Docker Container Registry | Amazon Web Services.** (n.d.). *Amazon Web Services, Inc.* https://aws.amazon.com/ecr/
- **AWS.** (2019). *AWS Lambda – Serverless Compute.* Amazon Web Services, Inc. https://aws.amazon.com/lambda/
- **Banerjee, S.** (2022, August 20). *House Rent Prediction Dataset.* Kaggle. https://www.kaggle.com/datasets/iamsouravbanerjee/house-rent-prediction-dataset
- **GDPR Implementation.** (2020). *Magicbricks.com.* https://www.magicbricks.com/
- **GitHub.** (2025). *GitHub.* https://github.com/
- **Keras.** (2019). *Home - Keras Documentation.* Keras.io. https://keras.io/
- **MLflow | MLflow.** (2025). *Mlflow.org.* https://www.mlflow.org/
- **Scikit-learn.** (2019). *scikit-learn: Machine learning in Python — scikit-learn 0.20.3 documentation.* Scikit-Learn.org. https://scikit-learn.org/stable/index.html
- **TensorFlow.** (2019). *TensorFlow.* Google. https://www.tensorflow.org/

Code Repository

- https://github.com/WorapolKhu/CPE393-group_name.git