

Containerized Machine Learning: House Price Predictor Project

Harsh Kumar

Reg. No: 12212246

School of Computer Science and
Engineering, Lovely Professional
University, Phagwara, Punjab, India
Email: hrk84ya@gmail.com

Abstract—This project implements a containerized machine learning microservice for predicting house prices using synthetic data. A Random Forest Regressor model is trained and served via a Flask API, with a minimal web interface for user interaction. The entire system is containerized using Docker and integrated into a Jenkins-based CI/CD pipeline for automated testing and deployment. Emphasizing modularity, scalability, and production-readiness, the project demonstrates how machine learning models can be effectively deployed as maintainable, real-time services.

Index Terms—Machine Learning, Flask API, Docker, Jenkins, CI/CD, Random Forest, Containerization.

I. Introduction

In today's software landscape, the demand for deploying machine learning (ML) models in production environments is growing rapidly. This paper presents an end-to-end pipeline for a house price prediction system developed as a containerized ML microservice. The project integrates synthetic data generation, model training, API deployment, containerization, and CI/CD automation to

ensure the ML service is scalable, portable, and production-ready.

Machine learning in production is no longer limited to enterprises. With the rise of MLOps and DevOps tools, individual developers and small teams can now build, test, and deploy reliable machine learning services. The goal of this project is to serve as a proof of concept that demonstrates the practical deployment of a real-time ML service in a modular, reproducible, and user-friendly manner.

II. Problem Statement and Motivation

Traditional ML workflows often lack production integration, resulting in models that are not used beyond the experimental phase. There exists a clear need for deploying ML models in a way that supports maintainability, scalability, and consistent performance across different environments.

This project aims to address this need by building an ML-powered web service that can make accurate predictions based on user input, and which can be deployed across various environments through containerization. The use of Jenkins for CI/CD further ensures code stability and automates testing and deployment.

III. System Architecture

The architecture follows a modular structure where each component serves a distinct purpose but integrates seamlessly with others.

- **Model Training Module:** Prepares synthetic data and trains the Random Forest Regressor model.
- **Flask API:** Hosts the web server and defines endpoints for prediction and health monitoring.
- **Frontend UI:** Built with HTML/CSS and JavaScript,

allowing users to input housing parameters and receive real-time predictions.

- **Containerization Layer:** Docker ensures that the entire service can run in isolated environments.
- **CI/CD Layer:** Jenkins handles automated testing and deployment, promoting continuous delivery.

This structure promotes modularity and makes the system scalable. New features, such as authentication or database support, can be added with minimal disruption.

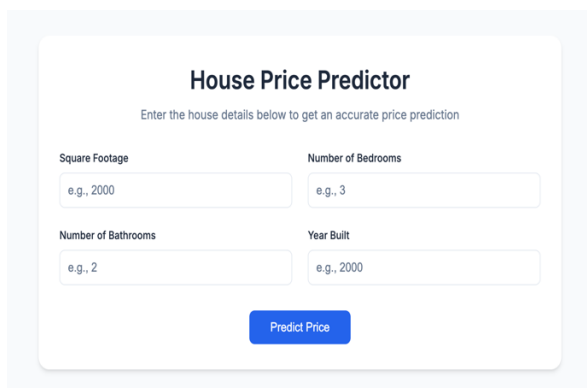


fig. UI of the House Price Predictor

IV. Data Generation and Preprocessing

Given the absence of real-world housing data, the project generates synthetic data to simulate realistic features:

- **Square footage:** Ranges from 1,000 to 5,000 sq ft
- **Bedrooms:** 1 to 5
- **Bathrooms:** 1 to 3
- **Year built:** Between 1960 and 2022

The house price is calculated using a weighted formula based on these features, with Gaussian noise added to simulate real-world variability. Feature scaling is performed using Standard Scaler to normalize the input space, and the dataset is split 80/20 for training and testing.

V. Model Development

The model development process focuses on achieving a balance between accuracy, performance, and interpretability.

- **Model Selection:** Random Forest Regressor was chosen for its robustness and ability to handle both numerical and categorical features.
- **Training Results:** The model achieved an R^2 of ~ 0.993 on training data and ~ 0.936 on testing data.
- **Persistence:** The model and scaler are stored together using pickle to ensure consistent preprocessing during prediction.

The model is resilient to overfitting due to ensemble learning and demonstrates good generalization performance.

VI. API and Frontend Development

The Flask application serves as the bridge between the trained model and the end user.

- **Endpoints:**
 - `/`: Serves HTML interface
 - `/predict`: Accepts JSON input for prediction
 - `/health`: Provides system status

The frontend is built using HTML and inline CSS. JavaScript handles form validation and asynchronous fetch calls to the backend. Input fields include numeric validation, and the results are dynamically displayed.

VII. Testing Strategy

Automated tests are written using pytest to validate critical API endpoints. These include:

- Checking if the / route returns a valid HTML response.
- Verifying the /health endpoint returns a healthy status.
- Ensuring /predict responds correctly to valid and invalid input.

Testing is part of the Jenkins pipeline to prevent regressions.

VIII. Containerization with Docker

Docker is used to package the application into a portable environment:

- **Base Image:** `python:3.9-slim` for reduced size
- **Structure:** App files, model, and dependencies are included
- **Ports:** Exposes port 5050 for the Flask app

This allows consistent execution across different operating systems and platforms.

```

1 FROM python:3.9-slim
2
3 WORKDIR /app
4
5 COPY model.pkl app.py requirements.txt ./
6
7 RUN pip install -r requirements.txt
8
9 EXPOSE 5050
10
11 CMD ["python", "app.py"]

```

fig. Dockerfile

IX. CI/CD Pipeline with Jenkins

The Jenkinsfile defines a multi-stage pipeline:

- **Setup:** Creates a virtual environment and installs dependencies
- **Test:** Runs unit tests using `pytest`
- **Build:** Constructs a Docker image tagged with Jenkins build number
- **Deploy:** Conditional deployment only from the `main` branch

Each stage includes logging and error handling for traceability and resilience.

```

12
13 stages {
14     stage('Setup') {
15         steps {
16             sh '''
17                 which python3 || echo "Python3 not found"
18                 python3 -m venv venv
19                 . venv/bin/activate && pip install -r requirements.txt
20             '''
21         }
22     }
23
24     stage('Test') {
25         steps {
26             sh '. venv/bin/activate && python3 -m pytest'
27         }
28     }
29 }

```

fig. Jenkinsfile

X. Deployment Strategy

Deployment is currently performed locally but is structured for future expansion:

- **Manual Deployment:** Uses Docker CLI to build and run the app
- **Automated Deployment:** Jenkins handles deployment via the pipeline

The application is accessible via `localhost:5050` and can be migrated to cloud providers such as AWS or GCP.

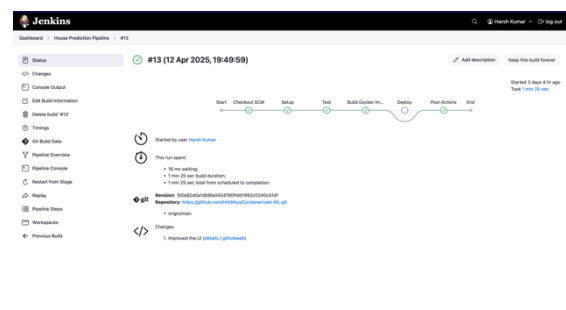


fig. Jenkins Pipeline

XI. Challenges and Solutions

Several technical issues were encountered:

- **File Access in Docker:** Resolved by updating Dockerfile paths
- **Model Inference Errors:** Solved by ensuring consistent data scaling
- **CI Failures:** Fixed Docker daemon access for Jenkins agents

- **Frontend Issues:** Enhanced error messages and validation for better UX

These were addressed by adhering to best practices in containerization and application design.

XII. Future Enhancements

Planned improvements include:

- Switching to real housing datasets (e.g., Ames or Zillow API)
- Adding model monitoring tools and automated retraining pipelines
- Implementing authentication and rate limiting for security
- Migrating to Kubernetes for better scaling and orchestration
- Enhancing frontend using modern frameworks like React or Vue

These enhancements aim to transform the project into a more robust, production-grade system.

XIII. Conclusion

This project demonstrates how machine learning can be deployed as a reliable microservice using modern DevOps practices. Through modular design, containerization, and automation, the system is capable of serving real-time predictions in a scalable, reproducible way. It provides a valuable blueprint for building ML-powered services ready for real-world deployment.

References

- [1] Scikit-learn documentation.
- [2] Flask documentation.
- [3] Docker official guide.
- [4] Jenkins pipeline documentation.
- [5] Machine Learning Handbook

GitHub:

<https://github.com/Hrk84ya/Containerized-Machine-Learning-House-Price-Predictor>

Docker Hub:

<https://hub.docker.com/r/hrk84ya/my-ml-app/tags>