Predicting NYC Taxi Trip Fares Using Machine Learning

Udyavara Narsim Kamath

Student ID – 14472885

Course: DATA_SCI 8420

Date of Submission: 12/17/2025

Predicting NYC Taxi Trip Fares Using Machine Learning Models on a High-Performance Computing Environment.

Abstract:

The project focuses on predicting New York City taxi trip fares using publicly available data from the New York City Taxi and Limousine Commission (TLC). Accurate fare estimation is crucial for passengers, taxi operators, and urban planners, requiring analysis of large-scale datasets with temporal, spatial, and trip-specific attributes. Some of the information included in the dataset is trip distance and duration, pricing rates, payment types, surcharges, and fare amounts.

To enhance the quality of the data, extensive preprocessing was performed to remove invalid and noisy records. Precautionary steps were taken to prevent data leakage, either by removing fare-related attributes or replacing them with indicator flags. Additionally, temporal and spatial features were engineered to capture more meaningful travel patterns.

Three machine learning models were used: Linear Regression (Lasso), Random Forest, and XGBoost. These models were trained and then evaluated using Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and the R-squared metric. XGBoost achieved the best overall performance, demonstrating its ability to capture non-linear relationships.

To support scalability and reproducibility, the project was deployed on the Nautilus High-Performance Computing (HPC) cluster using Kubernetes and Docker. This setup allowed for efficient resource utilization, parallel job execution, and reproducible experimentation. Overall, this project showcases the practical benefits of combining large-scale machine learning pipelines with HPC infrastructure.

Introduction:

Motivation:

Taxi fares in New York City are influenced by multiple factors, such as trip distance and duration, pickup and drop-off locations, traffic conditions, and surcharges. Accurately estimating these fares is not only important to passengers, who benefit from transparency and predictable pricing, but also for taxi operators, who can anticipate earnings, and city planners, who can analyze fare patterns to make informed transportation-based decisions. For a data scientist, the NYC TLC trip records provide a large-scale, diverse dataset with millions of trips, making it ideal for predicting fares using machine learning. Additionally, the dataset presents a variety of complex challenges, which, once addressed, help improve skills related to data cleaning, feature engineering, and computation.

Problem Statement:

This project aims to predict NYC yellow taxi fares using historical data from the New York City Taxi and Limousine Commission (NYC TLC). The goal is to estimate fares accurately based on features such as location, distance, and time.

The dataset is large and contains noisy and irregular trips, with complex non-linear relationships between features and fare amounts. To handle such data, the project implements machine learning models within a robust and reproducible workflow on an HPC cluster.

Objectives:

1) Developing machine learning models that are capable of accurately estimating fare amounts using diverse attributes.
2) Performing extensive preprocessing and feature engineering.
3) Identifying influential predictor variables.

4) Training and comparing the performance of different machine learning models.
5) Building a resource-efficient workflow on an HPC cluster using Kubernetes and Docker.

Contribution and Novelty:

While taxi fare predictions have been studied extensively in prior works, this project sets itself apart from the rest for the following reasons:

1) Machine learning pipeline on an HPC infrastructure: The project demonstrates how data ingestion, preprocessing, model training, and evaluation can be conducted on an HPC environment, providing enhanced scalability while also laying the groundwork for handling significantly larger datasets in the future.
2) Enhanced Reproducibility: The project makes use of a custom image containerized in Docker, along with Kubernetes pods and jobs, all of which are lightweight and portable. This allows for easy storing and sharing using a public repository in GitLab.

Related Work & Literature Review:

As already mentioned, taxi fare predictions have been studied extensively in prior work, especially with large datasets such as the New York City Taxi and Limousine Commission (NYC TLC) trip records. These works tend to explore a variety of preprocessing approaches along with different machine learning models to handle complexity, noise, and scale of the dataset.

Some of the earlier approaches relied heavily on Linear Regression, considering only trip distance, duration, and a few temporal features. They were easy to interpret, and computationally efficient, but underperformed due to their inability to capture non-linear relationships between spatial, temporal, and trip-specific variables.

To address these limitations, tree-based ensemble models such as Random Forest and XGBoost were widely used. Huang (2023) [2] highlighted that Random Forest consistently outperformed Linear Regression by modeling complex feature interactions. The study emphasized preprocessing, which included the removal of negative fares and unrealistic distances, and calculating geographic distances. Moorthy (2025) [1] provides a more recent evaluation on XGBoost under noisy conditions. This paper, like the one before, focused more on the importance of preprocessing by demonstrating how the removal of invalid trips and the imputation of missing values greatly enhanced predictive accuracy. Chou et al. (2023) [4] also confirmed that ensemble models tend to outperform linear approaches when working with heterogeneous spatial and temporal features. As for preprocessing, normalization, encoding, and outlier removal were mentioned as being critical to performance.

A few applied projects support these conclusions. Errhythm's GitHub project (2023) [3] incorporated Linear Regression, Random Forest, and XGBoost along with feature engineering, removal of invalid records and strict train/test splits to avoid data leakage. Similarly, a tutorial from Medium demonstrated these models with a strong emphasis on preprocessing, feature creation, and exclusion of target-related features such as surcharges, airport fees, tolls, and congestion fees to prevent leakage.

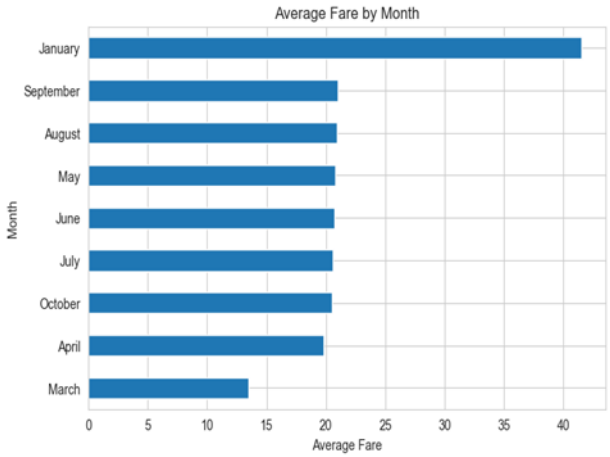
Aside from the machine learning approach, all prior studies emphasized preprocessing as a critical component. Some of the best practices included the removal of negative fares, invalid records, and outliers, creating meaningful spatial and temporal features, along with strict training and testing splits.

That being said, most of the prior work has been conducted on a single machine or small-scale environment, often relying on a subset of the data. They generally focus on model choice, preprocessing, and handling noisy trips but rarely discuss scalability and reproducibility.

This project builds on the prior work by deploying machine learning workflows for fare prediction on an HPC cluster using Kubernetes and Docker, enabling scalable training on large, feature-engineered datasets, along with efficient resource utilization and experiment reproducibility.

Methodology and Architecture:

The dataset used in this project comes from the New York City Taxi and Limousine Commission (NYC TLC) trip records, with a primary focus on yellow taxi trips. The data files were available up to October 2025, and as such, do not contain taxi trips for November and December.



Interestingly, there appears to be no data for the month of February. Possible explanations include delayed data collection in that time period or loss of records during preprocessing.

From the exploratory analysis (EDA), aside from January, most months exhibit similar average fare amounts, generally ranging from $20 to $21 per trip. As a result, month as a feature contributes limited predictive power when compared to other features available in the dataset.

Some of the information included in the dataset is as follows:

1) Pickup and drop-off timestamps and locations
2) Trip distance
3) Surcharges
4) Fare amount

The dataset originally comes in Parquet format, with each file being approximately 65 MB, totaling around 3GB after being converted to CSV format. After preprocessing and feature engineering, the final dataset size is approximately 1.5 GB, which is currently stored in the project's Persistent Volume Claim (PVC) under the name "**ukgff-finalproj**" on the HPC cluster.

Preprocessing steps:

1) Removal of invalid records: Trips that include negative fares or surcharges, unknown pickup/drop-off locations, unknown rate codes, zero or negative trip distance and duration, and zero passengers.
2) Feature engineering:
   - Temporal features (categorical)– hour of day, day of week, and month
   - Spatial features (categorical)– pickup and drop-off boroughs
3) Removal of unnecessary payment types: Only trips that were paid for with cash or card were retained, as these are the only ones that contribute to fare prediction.
4) Removal of unnecessary attributes: Attributes such as "store_and_fwd_flag" were removed as they were mostly meant for audit purposes.
5) Handling outliers:
   - Trips with fares in the $3-$500 range were retained.
   - Trips with a duration of 5-100 minutes were retained.
   - Trips with a distance greater than 0 but under 100 miles were retained.

6) Data leakage prevention: Removed features related to fare components (total amount, surcharges). Some were replaced with flags to highlight if toll roads were taken, the trip was to the airport, etc. Data was split into a training and testing set using an 80/20 split.

Dataset Statistics:

- Number of entries: 15 million
- Number of features: 19
- Target variable: fare_amount (Continuous variable)

Technical Approach:

Three machine learning models were used, and are as follows:

1) Linear Regression (with Lasso)
2) Random Forest
3) XGBoost

Hyperparameter tuning: A few parameters for XGBoost and Random Forest have been adjusted to improve model performance and generalization.

Evaluation metrics: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R-squared

Cloud Implementation:

Container Specification:

- Base Image – python:3.10-slim
- Dependencies: Pandas, NumPy, scikit-learn, XGBoost, joblib, and matplotlib

Resource allocation:

- CPU – 4 Cores
- RAM – 16 GB (8 GB was not sufficient for model training)
- GPU – Not Required

Kubernetes overview:

Kubernetes is used to run batch jobs for splitting data, model training, and evaluating model performance on unseen test data. Each job runs a Python script and uses a custom image. Each job mounts the PVC at /work_env, providing access to the dataset, scripts, models, and output directory. The restart policy was set to “never” to avoid reruns of completed jobs.

Data pipeline architecture:

1. Load clean, feature-engineered data from PVC
2. Split data into training and testing sets.
3. Train models on training sets and save trained models in PVC
4. Load trained models from PVC and evaluate performance. Results are stored in the PVC.

Job scheduling strategy:

Training jobs for the three models were run together in parallel, followed by the evaluation jobs. Kubernetes managed execution, enforced resource limits, and provided logs for monitoring purposes.

Implementation Details:

The project code is managed and organized in my public GitLab repository.

Link: https://gitlab.nrp-nautilus.io/N4e-mizzou/final-project-nyc_fare_prediction

For this project, the repository is broken into seven main directories to ensure clarity and reproducibility. These directories are as follows:

1) Documentation: Contains Jupyter notebooks for data cleaning, feature engineering, and EDA of the processed data, along with instructions to run and set up the project.
2) Scripts: Contains Python scripts for data splitting, model training, and performance evaluation.
3) Jobs: Contains YAML job files that execute Python scripts on Nautilus using a custom Docker image, with the project PVC (ukgff-finalproj) mounted.
4) Results: Contains evaluation outputs which include PNG visualizations and text performance metrics.
5) POD_PVC: Contains the YAML files used to create the project PVC and a pod to access the PVC.
6) Image: Contains the Docker, YAML, and text files required to build the custom Docker image.
7) Data: Contains raw data files used to construct the final dataset.

The dataset was pulled from my public Hugging Face data repository.

Link: https://huggingface.co/datasets/N4e-mizzou/NYC_Taxi_data

Containerization:

The project uses a custom Docker image created in GitLab.

Link: https://gitlab.nrp-nautilus.io/N4e-mizzou/ml_image

It is a lightweight, portable Docker image based on python:3.10-slim. The dependencies installed include Pandas, NumPy, scikit-learn, XGBoost, joblib, and matplotlib, ensuring a reproducible and consistent environment across different HPC clusters.

The Docker image was built in GitLab using the files in the “Image” directory of the project repository and then pushed into the container registry. For job execution, the image path is specified in the image field of the YAML job files.

Image Paths:

- gitlab-registry.nrp-nautilus.io/n4e-mizzou/ml_image:428120b1
- gitlab-registry.nrp-nautilus.io/n4e-mizzou/ml_image:latest

Nautilus Setup:

Access to Nautilus was established via the JupyterHub interface using CILogon, which enables secure login with institutional credentials. For direct access to the cluster, an NRP account was created, followed by a request for access to the namespace gp-engine-mizzou-dsa-cloud.

Additionally, the Kubernetes configuration file was downloaded from the NRP portal, which was then modified and deployed to the .kube/config location, allowing the JupyterHub environment to communicate with the cluster. To interact with the cluster, kubectl command-line tools were used.

A PVC (ukgff-finalproj) is configured with ReadWriteMany access and a storage request of 5 GB to store the dataset, trained models, and evaluation results. The PVC was mounted as a shared read/write volume, which allows all jobs to access the same data and outputs concurrently.

The commands used were as follows:

- kubectl -n gp-engine-mizzou-dsa-cloud create -f <YAML file> (Create a job, pod, or PVC)
- kubectl -n gp-engine-mizzou-dsa-cloud get jobs/pods/pvc (Monitor the status of jobs, pods, and PVCs)
- kubectl -n gp-engine-mizzou-dsa-cloud logs <pod/job name> (Retrieves logs for debugging and verification)
- kubectl -n gp-engine-mizzou-dsa-cloud describe job/pod <job/pod name> (Shows detailed information about the Kubernetes resource)

Execution:

1) Pull the project repository from the GitLab link provided above.
2) Create the custom image in a separate repository with the files provided in the Image directory. Alternatively, the image paths provided above can also be used.
3) Create a PVC using the persistent_volume.yml file provided in the POD_PVC directory.
4) Create a pod using the pod_pvc.yml file provided in the POD_PVC directory. Using the created pod, access the PVC.
5) Within the /work_env directory of the PVC, create /cleandata, /result, and /Scripts directories.
6) Within the /cleandata directory, pull the dataset from the Hugging Face link provided above.
7) Exit the PVC.
8) Copy the scripts from the /Scripts directory on JupyterHub into the /Scripts directory of the project PVC.
9) Run the Data_split.yaml job. Wait until completion.
10) Run RF.yaml, LR_Lasso.yaml, and XGB.yaml jobs. Wait until completion.
11) Run Test_LR.yaml, Test_RF.yaml, and Test_XGB.yaml jobs. Wait until completion.
12) Use the same pod as before to access the PVC. Within the /work_env/result directory, visuals of performance evaluations on test data can be found.
13) Use the kubectl logs <jobname> command for the jobs created in step 10 to view performance metrics.
14) Additionally, EDA_XGB.yaml, EDA_Test_XGB.yaml jobs can be run one after the other to evaluate the performance of the best performing model on a selected set of features.

Challenges and Solutions:

The only challenge I faced was figuring out how to load my large, clean, and feature-engineered dataset from my local machine into the PVC. Since direct upload was not an option, I stored the dataset in a public Hugging Face repository and securely pulled it into the PVC using a read-only API key. Aside from this, no major technical challenges were encountered.

Results:

Execution time of each model on Nautilus is as follows:

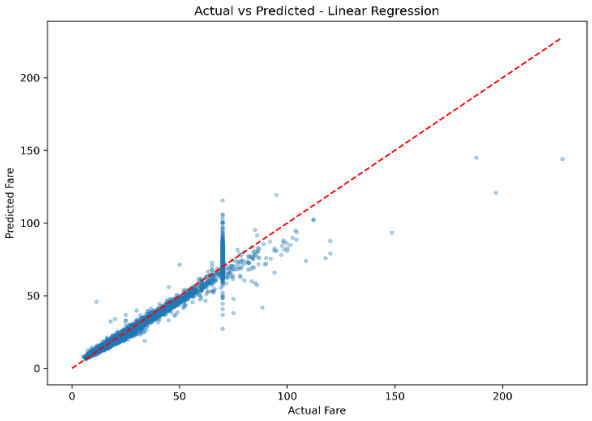
1) Linear Regression:
    - Model training: 3 minutes and 23 seconds
    - Model evaluation: 24 seconds

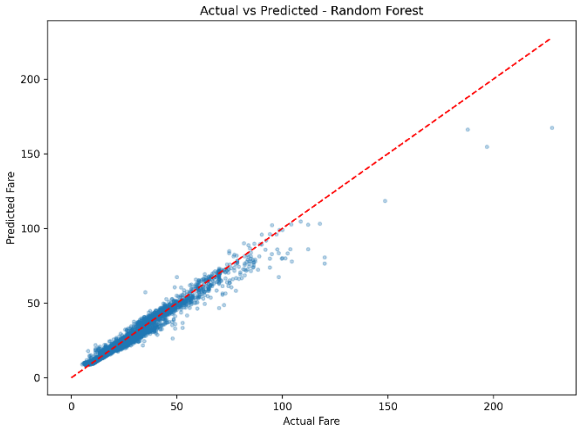
2) Random Forest:
    - Model training: 13 minutes
    - Model evaluation: 43 seconds
3) XGBoost
    - Model training: 4 minutes and 52 seconds
    - Model evaluation: 38 seconds

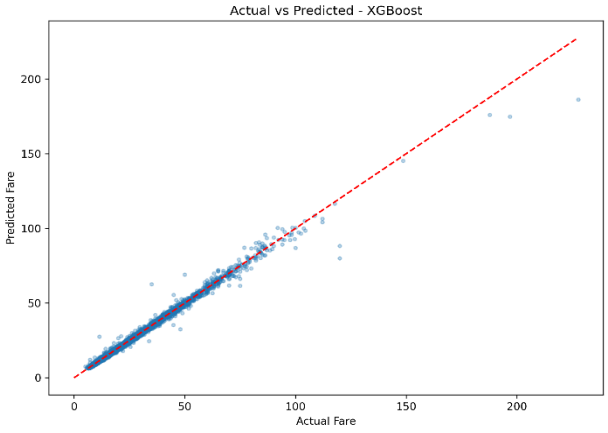
All jobs used 4 CPU cores and 16 GB RAM to execute. Initially, 8 GB RAM was used, however, after the 2-minute mark, the jobs would fail due to insufficient memory.

The performance metrics of each model are as follows:

1) Linear Regression:
    - MAE: 1.57
    - RMSE: 3.40
    - R^2: 0.9596
2) Random Forest:
    - MAE: 1.33
    - RMSE: 2.57
    - R^2: 0.9770
3) XGBoost:
    - MAE: 0.53
    - RMSE: 1.51
    - R^2: 0.9921







Across all three models, training and test performance are consistent, indicating strong generalization and minimal overfitting.

Linear Regression (Lasso) serves as a strong baseline, confirming that NYC taxi trips are driven by features such as distance and time. Random Forest further improves performance by capturing non-linear relationships while also reducing error. XGBoost performs the best overall on both training and testing data, cementing itself as the final model choice.

From EDA, it was found that trip distance and duration, trip type, toll roads used, and pickup/drop-off locations are primary factors influencing fare amounts. To confirm this, XGBoost was trained using only these features, achieving the following performance metrics:

- MAE: 0.57

- RMSE: 1.57
- R^2: 0.9914

The performance is remarkably close to the model trained on all features, indicating that the predictors identified during EDA capture most of the information needed for accurately predicting NYC taxi fares.

Reproducibility:

All the results are fully reproducible using the provided GitLab repository, Docker image, and job configurations. Within each Python script, a fixed random seed (seed = 42) was set to ensure predictable behavior. Using the same container environment guarantees consistent library versions and executes jobs in an identical environment, preventing variability in results.

Discussion and Analysis:

The results show that trip distance and duration, trip type, toll roads used, and pickup/drop-off locations are strong predictors of NYC taxi fares, with XGBoost achieving the highest accuracy.

When talking about limitations, they come in two ways:

- Data: The dataset contains a significant amount of noisy trips, meaning that some records have unusual or inconsistent values such as negative fares, zero or extreme durations, and implausible distances. These records must be removed to ensure accurate predictions, but in doing so, a substantial amount of potentially informative data is lost. Due to the preprocessing of the data, the number of entries was reduced from 25 million to 14 million.
- Resource: Each job is limited to the number of CPU cores and RAM allocated. If a model requires more memory than is available, it will fail. Therefore, careful planning is needed to ensure successful job completion while efficiently managing CPU and memory usage.

Despite these limitations, the Nautilus HPC cluster improves the scalability of workflows to larger datasets and more complex models by adjusting resource allocation and allowing parallel job scheduling. The deployment process could be improved by automating job scheduling and dynamically allocating resources. Overall, this project has provided me with the practical experience in working with containerized environments, orchestrating Kubernetes jobs, pods, and PVCs and building a reproducible HPC-based machine learning pipeline.

Conclusion:

This project developed a scalable and reproducible machine learning pipeline to predict NYC taxi fare using large-scale historical data. Feature engineering and EDA identified trip distance and duration, trip type, toll roads used, and pickup/drop-off locations as primary factors influencing fare amount, with XGBoost achieving the best overall performance. The use of Docker and Kubernetes on an HPC cluster allowed for efficient resource utilization, parallel execution, and reproducibility.

Future work could explore more complex models like CatBoost and LightGBM with extensive hyperparameter tuning. Additionally, dynamic resource allocation and automated job scheduling could be looked into to further improve deployment efficiency.

Overall, this project provided not only practical experience but also demonstrated the benefits of HPC technology for large-scale machine learning pipelines.

References:

[1] Moorthy, P. (2025). *Robust Taxi Fare Prediction Under Noisy Conditions: A Comparative Study of GAT, TimesNet, and XGBoost*. arXiv eprint arXiv:2507.20008., Link: https://doi.org/10.48550/arXiv.2507.20008

[2] Huang, H. (2023). *Taxi fare prediction based on multiple machine learning models*. Applied and Computational Engineering, 16, 2755-2721., Link: https://doi.org/10.54254/2755-2721/16/20230849

[3] Errhythm (2023). *NYCTaxiFarePred-Extended*. GitHub., Link: https://github.com/errhythm/NYCTaxiFarePred-Extended

[4] Chou, K. S., Wong, K. L., Zhang, B., Aguiari, D., Im, S. K., Lam, C. T., Tse, R., Tang, S.-K., & Pau, G. (2023). *Taxi Demand and Fare Prediction with Hybrid Models: Enhancing Efficiency and User Experience in City Transportation*. Applied Sciences, *13*(18), 10192., Link: https://doi.org/10.3390/app131810192

[5] Medium (2023). *Predicting Taxi Fare Prices using XGBoost and Random Forest*., Link: https://medium.com/@rahil4088/predicting-taxi-fare-prices-using-xgboost-and-random-forest-fe1ed48441db

[6] New York City Taxi and Limousine Commission. TLC trip record data. NYC.gov. (2025)., Link: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[7] McKinney, W. (2010). *Data Structures for Statistical Computing in Python*. Proceedings of the 9th Python in Science Conference (SciPy 2010), 51-56., Link: https://pandas.pydata.org/

[8] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). *Array programming with NumPy*. Nature, 585(7825), 357-362., Link: https://numpy.org/

[9] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. (2011). *Scikit-learn: Machine learning in Python*. Journal of Machine Learning Research, 12, 2825–2830., Link: https://scikit-learn.org/stable/

[10] Chen, T., & Guestrin, C. (2016). *XGBoost: A scalable tree boosting system*. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 785–794., Link: https://xgboost.ai/

[11] The joblib developers. (2023). joblib (1.4.2). *Joblib: running Python functions as pipeline jobs*. Zenodo., Link: https://joblib.readthedocs.io/en/stable/

[12] Hunter, J. D. (2007). *Matplotlib: A 2D graphics environment*. Computing in Science & Engineering, 9(3), 90–95., Link: https://matplotlib.org/