FarePrediction AI: In-Depth Project Documentation

Author: Doolam Rohith Goud
Project: An Explainable AI (XAI) System for NYC Taxi Fare Prediction

Part 1: The Initial Idea & Goal

The project's foundation is a common problem in the modern digital economy: the "black box" algorithm. Users of ride-sharing platforms are given a price, but the factors influencing that price are hidden. This lack of transparency can erode user trust and make it difficult to understand fare variability.

The primary goal was to architect an end-to-end machine learning system that not only provides accurate fare predictions for NYC taxis but also demystifies the prediction process for the end-user.

This objective led to the core concept of "Fair Fare AI." The system was designed to deliver a clear, itemized breakdown of the estimated cost. By showing exactly how factors like trip distance, time of day, and day of the week contribute to the final price, we aimed to build a more transparent and trustworthy user experience.

Part 2: The Machine Learning Model (The "Brain")

This phase, conducted within a Jupyter Notebook (nyc-taxi-fare-prediction-filled.ipynb), was dedicated to building the predictive engine. The process followed a standard data science workflow using Python and its ecosystem of libraries (Pandas, Scikit-learn, XGBoost, etc.).

1. Data Collection & Cleaning:

- **Source:** We utilized the "NYC Taxi Fare Prediction" dataset from Kaggle, a large-scale, real-world dataset.
- **Initial State:** The raw dataset contained over 55 million ride records. For efficient development, we worked with a representative sample.
- **Data Cleaning:** Before modeling, a crucial data cleaning step was performed. This involved removing outliers and invalid data points, such as:
  - Rides with a fare amount less than the legal minimum (e.g., < $2.50) or excessively high fares.
  - Rides with invalid latitude/longitude coordinates that placed them far outside the New York City area.
  - Rides with invalid passenger counts (e.g., 0 or more than 8).

2. Feature Engineering:

- This was the most critical step for model performance. We transformed raw data

into meaningful features that the model could learn from:
  - **Haversine Distance:** We calculated the great-circle distance between pickup and dropoff coordinates. This is the correct method for calculating distances on a sphere (the Earth) and is far more accurate than simple Euclidean distance for this use case. This feature had the highest predictive power.
  - **Time-Based Features:** The pickup_datetime timestamp was decomposed into several cyclical and seasonal features: year, month, day, weekday, and hour. This allows the model to learn complex temporal patterns, such as higher demand during rush hours (e.g., 8 AM, 5 PM) or on weekends.
  - **Landmark Proximity:** We engineered features representing the distance from the dropoff point to key NYC landmarks like JFK Airport, LaGuardia Airport, and Times Square. This helps the model capture location-based pricing dynamics, such as airport surcharges.

3. Model Selection & Training:

- **Algorithm:** We selected the **XGBoost (Extreme Gradient Boosting)** algorithm. It was chosen for its proven high performance on structured (tabular) data, its ability to handle complex interactions between features, and its computational efficiency.
- **Training Process:** The model was trained on the cleaned, engineered dataset. It learned to map the input features (distance, time, location, etc.) to the target variable (fare_amount) by iteratively building a series of decision trees, with each new tree correcting the errors of the previous ones.

4. Explainable AI (XAI) Integration:

- To achieve our core goal of transparency, we integrated the **SHAP (SHapley Additive exPlanations)** library.
- **Mechanism:** After training the XGBoost model, we created a SHAP TreeExplainer. This tool uses principles from cooperative game theory to assign a precise contribution value (a "SHAP value") to each feature for any given prediction.
- **Output:** For a single ride, the SHAP explainer tells us the model's average prediction (the "base value") and then shows how each feature (e.g., trip_distance = +$12.50, pickup_hour = +$2.10) pushed the final prediction away from that average. This became the technical foundation for our transparent fare breakdown.

Part 3: The Full-Stack Application (The "Body")

This phase involved operationalizing the trained model by building a professional web application around it.

1. Exporting the Model:

- The trained XGBoost model and the SHAP explainer, which are Python objects, were serialized using the joblib library. This process converts the in-memory objects into a file format (.pkl) that can be saved to disk and loaded later. This file, **xgb_model_final.pkl**, is the portable "brain" of our application.

2. Building the Backend (The "Engine"):

- **Framework:** We used **Flask**, a lightweight Python web framework, to create our server. The backend code is contained in the **app.py** file.
- **REST API:** We designed a RESTful API with a primary endpoint (/predict). This API acts as a clean, stateless interface between the frontend and the complex ML logic, following standard client-server architecture.
- **Geocoding Service:** To enable a user-friendly interface where users can type location names, we integrated the free **Nominatim** geocoding API (based on OpenStreetMap). When the backend receives a text-based location like "Empire State Building," it sends a request to this external API to get the precise latitude and longitude.
- **Core Logic:** The backend orchestrates the entire prediction process:
  1. Receives a POST request from the user's browser containing pickup/dropoff names.
  2. Calls the geocoding service to convert these names to coordinates.
  3. Executes the feature engineering pipeline on the coordinates.
  4. Feeds the engineered features into the loaded XGBoost model to get a prediction.
  5. Uses the SHAP explainer to calculate the detailed fare breakdown.
  6. Sends the final, structured result back to the browser in **JSON** (JavaScript Object Notation) format.

3. Creating the Frontend (The "Face"):

- The user interface is a single **index.html** file located in a templates folder, which Flask is configured to serve.
- **Structure & Style (HTML/CSS):** We used HTML to define the structure of the page (input fields, buttons, results area) and modern CSS to create a clean, professional "light mode" theme with high contrast for readability.
- **Interactivity (JavaScript):** The frontend's dynamic behavior is powered by JavaScript:
  1. It listens for the "Estimate Fare" button submit event.
  2. It uses the fetch API to make an asynchronous POST request to our Flask backend's /predict endpoint, sending the user's input as a JSON payload.

3. Upon receiving the JSON response from the backend, it dynamically manipulates the DOM (Document Object Model) to render the transparent fare breakdown in the results card, ensuring all components perfectly add up to the final total.

Part 4: Running the Application Locally

This is the final phase, where we execute the application on a local development server.

1. Project Setup:

- A main project folder, FairFarePredictor, was created to house all application components.
- Inside, we placed app.py (the server) and xgb_model_final.pkl (the model).
- A subfolder named templates was created to hold the index.html file, a standard Flask convention.

2. Installing Dependencies:

- We used the **Command Prompt** to navigate into the project folder.
- We ran the command pip install -r requirements.txt (or a direct pip install ... command) to install all the necessary Python libraries (Flask, joblib, pandas, numpy, shap, xgboost, requests). pip is Python's package manager.

3. Starting the Server:

- In the Command Prompt, we executed the command python app.py. This runs the Python script, which in turn starts the Flask development server.
- The server began listening for incoming web requests at the local address http://127.0.0.1:5000. 127.0.0.1 is a special IP address that always means "this computer."

4. Viewing the Application:

- We opened a web browser and navigated to our application's specific URL: **http://127.0.0.1:5000/fare-prediction-nyc**.
- This sent a GET request to our running Flask server, which responded by rendering the index.html page, displaying our final, professional web application.