**Car Maintenance Cost Prediction: Project Documentation**

**Author: Sujal Patidar**

**Table of Contents**

1. **Chapter 1: Project Overview**
   * 1.1. Introduction
   * 1.2. Problem Statement
   * 1.3. Project Objectives
   * 1.4. Technology Stack
   * 1.5. Project Directory Structure
2. **Chapter 2: Exploratory Data Analysis (EDA)**
   * 2.1. Introduction to EDA
   * 2.2. Data Loading and Initial Inspection
   * 2.3. Data Cleaning and Preprocessing in the Notebook
   * 2.4. Univariate Analysis (Visualizing Features)
     + 2.4.1. Categorical Features
     + 2.4.2. Numerical Features
   * 2.5. Key Insights from EDA
3. **Chapter 3: Model Training and Experimentation**
   * 3.1. Introduction to Model Training
   * 3.2. Data Preparation for Modelling
   * 3.3. Model Selection
   * 3.4. Model Evaluation Metrics
   * 3.5. Experimentation Results
   * 3.6. Final Model Selection
4. **Chapter 4: Modular Code Implementation (The src Directory)**
   * 4.1. The Importance of Modular Programming
   * 4.2. Core Utilities
     + 4.2.1. logger.py: The Logging Framework
     + 4.2.2. exception.py: Custom Exception Handling
     + 4.2.3. utils.py: Helper Functions
   * 4.3. The ML Components (src/components/)
     + 4.3.1. data\_ingestion.py: Reading and Splitting the Data
     + 4.3.2. data\_transformation.py: The Preprocessing Pipeline
     + 4.3.3. model\_trainer.py: Automated Model Training and Selection
   * 4.4. The Prediction Pipeline (src/pipeline/)
     + 4.4.1. predict\_pipeline.py: Orchestrating Predictions
5. **Chapter 5: Web Application and Deployment**
   * 5.1. Introduction to Flask
   * 5.2. app.py: The Flask Application
   * 5.3. templates/: The User Interface
     + 5.3.1. index.html
     + 5.3.2. home.html
6. **Chapter 6: Setup and How to Run**
   * 6.1. Project Dependencies (requirements.txt)
   * 6.2. Making the Project Installable (setup.py)
   * 6.3. Step-by-Step Execution Guide

**Chapter 1: Project Overview**

**1.1. Introduction**

The "Car Maintenance Cost Prediction" project is an end-to-end machine learning application designed to predict the maintenance cost of a vehicle based on various features. These features include the car's brand, model, make year, mileage, region, and a detailed list of services performed.

This project demonstrates a complete ML lifecycle, from initial data exploration and model experimentation in Jupyter notebooks to building a robust, modular, and deployable web application using Flask.

**1.2. Problem Statement**

For both car owners and service centers, accurately estimating maintenance costs can be challenging. Costs vary significantly based on the car's age, model, mileage, and the specific services required. An automated system that can provide a reliable cost estimate can be highly beneficial. It empowers car owners to budget effectively and helps service centers provide transparent and consistent pricing.

This project aims to solve this problem by building a regression model that predicts the cost of vehicle maintenance.

**1.3. Project Objectives**

* Perform a thorough Exploratory Data Analysis (EDA) to understand the data, identify patterns, and uncover relationships between features.
* Preprocess the data to make it suitable for machine learning models.
* Train, evaluate, and compare multiple regression models to find the one with the best predictive performance.
* Structure the project code in a modular and scalable way, following industry best practices.
* Develop a user-friendly web interface using Flask where users can input vehicle details and get a predicted maintenance cost.
* Implement robust logging and exception handling for easier debugging and maintenance.

**1.4. Technology Stack**

The project utilizes a range of Python libraries and frameworks:

* **Data Analysis & Manipulation**: Pandas, NumPy
* **Data Visualization**: Matplotlib, Seaborn
* **Machine Learning**: Scikit-learn
* **Advanced Models**: XGBoost, CatBoost
* **Web Framework**: Flask
* **Serialization**: Dill, Pickle
* **Development Environment**: Jupyter Notebook, VS Code

**1.5. Project Directory Structure**

The project is organized into a clean, standard structure for ML projects.

.

├── notebooks/

│ ├── EDA.ipynb # Exploratory Data Analysis notebook

│ └── MODEL\_TRAINING.ipynb # Model training experimentation notebook

├── src/

│ ├── components/

│ │ ├── data\_ingestion.py

│ │ ├── data\_transformation.py

│ │ └── model\_trainer.py

│ ├── pipeline/

│ │ └── predict\_pipeline.py

│ ├── \_\_init\_\_.py

│ ├── exception.py

│ ├── logger.py

│ └── utils.py

├── templates/

│ ├── home.html

│ └── index.html

├── app.py # Main Flask application file

├── requirements.txt # Project dependencies

└── setup.py # Makes the project installable

* **notebooks/**: Contains Jupyter notebooks used for initial research, EDA, and model prototyping.
* **src/**: The "source" directory containing all the core Python modules.
  + components/: Holds the main stages of the ML pipeline (ingestion, transformation, training).
  + pipeline/: Contains the prediction pipeline logic.
  + exception.py, logger.py, utils.py: Helper modules for handling errors, logging, and common functions.
* **templates/**: Contains HTML files for the Flask web application's user interface.
* **app.py**: The entry point for running the web application.
* **requirements.txt**: Lists all Python packages required to run the project.
* **setup.py**: A script to package the src directory, making the project's modules importable and distributable.

**Chapter 2: Exploratory Data Analysis (EDA)**

**2.1. Introduction to EDA**

EDA is the first and one of the most critical steps in any data science project. The goal is to analyze and investigate the dataset to summarize its main characteristics, often with visual methods. This helps us understand the data's structure, identify outliers, find patterns, and formulate hypotheses.

**2.2. Data Loading and Initial Inspection**

The dataset Vehicle Maintenance- Service Records.csv was loaded into a Pandas DataFrame.

* df.head(): Showed the first few rows, giving a feel for the columns and data types.
* df.info(): Revealed that the dataset has 1139 entries and 24 columns with no missing values. Most columns are int64, and five are object (categorical).
* df.describe(): Provided summary statistics for the numerical columns (mean, standard deviation, min, max, quartiles).

**2.3. Data Cleaning and Preprocessing in the Notebook**

Based on initial analysis, several columns were deemed irrelevant for predicting the maintenance cost and were dropped.

* **df.drop(['slno', 'oil\_filter', 'engine\_oil', 'vehicle\_type'], axis=1, inplace=True)**:
  + slno: A serial number that carries no predictive power.
  + vehicle\_type: This column contained only one unique value ('car'), making it redundant.
  + oil\_filter and engine\_oil: These features also contains only one unique value.

The features were then separated into numerical and categorical types for visualization.

**2.4. Univariate Analysis (Visualizing Features)**

**2.4.1. Categorical Features**

Count plots were generated for each categorical feature to understand their distribution (see EDA.ipynb, page 4).

* **Brand**: The dataset contains more **Honda** vehicles than **Toyota**
* **Model**: The models (jazz, amaze, city, fortuner) have a relatively balanced distribution, except for fortuner which has fewer entries.
* **Engine Type**: There are more diesel cars than petrol cars.
* **Region**: The data is split between mumbai and chennai, with slightly more entries from Mumbai.

**2.4.2. Numerical Features**

Histograms and boxplots were used to visualize the distribution of numerical features.

* **mileage and mileage\_range**: These features are roughly normally distributed but have several outliers on the higher end, indicating some vehicles with very high mileage.
* **Service Features** (e.g., drain, dust\_and\_pollen\_filter, air\_clean\_filter): These are binary (0 or 1) features. Their histograms show that for most services, the 'No' (0) case is far more common than the 'Yes' (1) case. This indicates that most maintenance jobs do not include all possible services.
* **cost (Target Variable)**: The distribution of cost is roughly normal but slightly right-skewed, with some high-cost outliers.

**2.5. Key Insights from EDA**

1. The dataset is clean with no missing values.
2. Several features are binary, representing whether a specific service was performed.
3. The target variable cost and some numerical features like mileage have outliers that the model needs to be robust against.
4. There are clear categorical distinctions (brand, model, region) that will likely be important predictors.

**Chapter 3: Model Training and Experimentation**

**3.1. Introduction to Model Training**

After exploring the data, the next step is to build and train machine learning models. The notebook MODEL\_TRAINING.ipynb serves as a "workbench" to experiment with different algorithms and find the most promising ones before formalizing the process in the modular scripts.

**3.2. Data Preparation for Modeling**

1. **Feature and Target Split**: The DataFrame was split into features (X) and the target variable (y, which is cost).
2. **Preprocessing with ColumnTransformer**:
   * **Numerical Features**: mileage\_range and mileage were scaled using StandardScaler. Scaling ensures that features with larger value ranges do not dominate the model's learning process.
   * **Categorical Features**: brand, model, engine\_type, make\_year, and region were transformed using OneHotEncoder. This converts categorical text data into a numerical format that models can understand, creating a new binary column for each category.
3. **Train-Test Split**: The preprocessed data was split into training (80%) and testing (20%) sets using train\_test\_split. This is crucial to evaluate the model's performance on unseen data.

**3.3. Model Selection**

A variety of standard regression models were chosen for experimentation to cover different modeling approaches:

* **Linear Models**: Linear Regression, Lasso, Ridge
* **Distance-Based Model**: K-Neighbors Regressor
* **Tree-Based Models**: Decision Tree, Random Forest
* **Ensemble/Boosting Models**: XGBoost, CatBoost, AdaBoost

**3.4. Model Evaluation Metrics**

The performance of each model was evaluated using the **R² Score (Coefficient of Determination)**.

* **R² Score**: This metric represents the proportion of the variance in the dependent variable (cost) that is predictable from the independent variables (the features). An R² score of 1 indicates a perfect prediction, while a score of 0 means the model performs no better than simply predicting the mean of the target. Higher is better.

**3.5. Experimentation Results**

The models were trained on the training data, and their R² scores were calculated on the test data. The results were as follows:

|  |  |
| --- | --- |
| Model Name | R2\_Score (Test Set) |
| **CatBoosting Regressor** | **0.8931** |
| K-Neighbors Regressor | 0.8862 |
| XGBRegressor | 0.8796 |
| Random Forest Regressor | 0.8644 |
| Decision Tree | 0.7993 |
| AdaBoost Regressor | 0.6526 |
| Lasso | 0.5759 |
| Linear Regression | 0.5759 |
| Ridge | 0.5757 |

**3.6. Final Model Selection**

From the results, it is clear that the tree-based ensemble models performed the best.

* **CatBoost Regressor** achieved the highest R² score of **0.8931**.
* K-Neighbors, XGBoost, and Random Forest also performed very well.
* The linear models (Linear Regression, Lasso, Ridge) performed poorly, indicating that the relationships in the data are likely non-linear and complex.

Based on this experimentation, **CatBoost Regressor** was selected as the best-performing model to be used in the final, automated pipeline. The high accuracy (89.31%) demonstrates its ability to effectively predict car maintenance costs from the given features.

**Chapter 4: Modular Code Implementation (The src Directory)**

**4.1. The Importance of Modular Programming**

While Jupyter notebooks are excellent for exploration, they are not ideal for building production-ready applications. Modular programming involves breaking down the project into smaller, independent, and reusable Python modules (.py files). This approach offers several advantages:

* **Maintainability**: Easier to fix bugs or update specific parts of the code.
* **Reusability**: Components can be reused across different projects.
* **Testability**: Individual modules can be tested in isolation.
* **Collaboration**: Multiple developers can work on different modules simultaneously.

**4.2. Core Utilities**

These are foundational scripts that support the entire application.

**4.2.1. logger.py: The Logging Framework**

This script sets up a custom logger.

* **Purpose**: Instead of using print() statements for debugging, logging provides a structured way to record events, warnings, and errors. This is essential for monitoring the application when it's running.
* **Functionality**:
  + It creates a logs directory if one doesn't exist.
  + It creates a log file named with the current timestamp (e.g., 10\_26\_2023\_14\_30\_00.log), ensuring each run has its own log file.
  + It configures the logging format to include the timestamp, line number, logger name, level (INFO, ERROR), and the message itself.

**4.2.2. exception.py: Custom Exception Handling**

This script defines a custom exception class, CustomException.

* **Purpose**: Standard Python exceptions can sometimes be vague. A custom exception allows us to catch errors and re-raise them with more context.
* **Functionality**:
  + The error\_message\_detail function extracts the file name and line number where an error occurred.
  + The CustomException class inherits from Python's base Exception class and formats a detailed error message, making debugging significantly easier. For example: Error occurred in python script name [src/components/data\_ingestion.py] line number [35] error message[...].

**4.2.3. utils.py: Helper Functions**

This module contains utility functions that are used by multiple components.

* **save\_obj(file\_path, obj)**: This function saves any Python object (like a model or a preprocessor) to a file using dill. dill is an advanced version of pickle that can serialize a wider range of Python objects.
* **evaluate\_models(...)**: This function automates the model training and evaluation process. It takes a dictionary of models and their hyperparameters, uses GridSearchCV to find the best parameters for each model, trains it, and returns a report of their R² scores.
* **load\_object(file\_path)**: This function loads a saved Python object from a file.

**4.3. The ML Components (src/components/)**

These modules represent the core stages of the machine learning training pipeline.

**4.3.1. data\_ingestion.py: Reading and Splitting the Data**

* **DataIngestionConfig**: A dataclass that defines the output paths for the raw, train, and test data files inside an artifacts folder.
* **DataIngestion class**:
  + initiate\_data\_ingestion(): This is the main method. It reads the raw CSV data, saves a copy, and then splits it into training and testing sets (80/20 split). These sets are saved as train.csv and test.csv in the artifacts folder. This ensures that the data split is consistent and reproducible.

**4.3.2. data\_transformation.py: The Preprocessing Pipeline**

* **DataTransformationConfig**: Defines the path where the preprocessing object will be saved.
* **DataTransformation class**:
  + get\_data\_transformer\_obj(): This method defines the entire preprocessing logic using ColumnTransformer and Pipeline. It specifies which columns are numerical and which are categorical, and defines the steps for each (e.g., SimpleImputer for missing values, StandardScaler for numerical, OneHotEncoder for categorical).
  + initiate\_data\_transformation(): This method takes the paths of the train and test CSV files, applies the preprocessing object to them, and returns the transformed data as NumPy arrays ready for model training. It also saves the fitted preprocessor object (preprocessor.pkl) to the artifacts folder. This is crucial because the same transformation must be applied to new data during prediction.

**4.3.3. model\_trainer.py: Automated Model Training and Selection**

* **ModelTrainerConfig**: Defines the path for saving the final trained model (model.pkl).
* **ModelTrainer class**:
  + initiate\_model\_trainer(): This is the heart of the training process.
    1. It takes the transformed train and test arrays.
    2. It defines a dictionary of models to be evaluated (Random Forest, XGBoost, etc.).
    3. It defines a dictionary of hyperparameters for GridSearchCV to search through.
    4. It calls the evaluate\_models function from utils.py to get a performance report.
    5. It identifies the best model based on the highest R² score.
    6. It saves the best-performing model object to artifacts/model.pkl using the save\_obj utility function.
    7. It returns the R² score of the best model.

**4.4. The Prediction Pipeline (src/pipeline/)**

This module handles the logic for making predictions on new, unseen data.

**4.4.1. predict\_pipeline.py: Orchestrating Predictions**

* **CustomData class**: This class is a bridge between the user input from the web form and the format required by the model. It takes all the input features (brand, model, mileage, etc.) and has a method get\_data\_as\_data\_frame() that organizes them into a Pandas DataFrame with the correct column names. This is exactly the format our preprocessor was trained on.
* **PredictPipeline class**:
  + predict(features): This method takes the DataFrame created by CustomData. It loads the saved preprocessor.pkl and model.pkl objects from the artifacts folder. It first transforms the input features using the loaded preprocessor and then feeds the transformed data to the loaded model to get the final cost prediction.

**Chapter 5: Web Application and Deployment**

**5.1. Introduction to Flask**

Flask is a lightweight and flexible Python web framework. It is used here to create a simple web server that can host our model and provide a user interface for interaction.

**5.2. app.py: The Flask Application**

This script is the entry point for running the application.

* application = Flask(\_\_name\_\_): Initializes the Flask app.
* **Routes**:
  + @app.route('/'): This is the index or landing page. It simply renders index.html.
  + @app.route('/predictdata', methods=['GET', 'POST']): This is the main prediction endpoint.
    - If the request method is GET, it displays the data entry form by rendering home.html.
    - If the request method is POST (meaning the user has filled the form and clicked "Predict"), it retrieves all the data from the form. It then uses the CustomData and PredictPipeline classes to process the inputs and get a prediction. Finally, it re-renders home.html but this time passes the prediction result to be displayed to the user.

**5.3. templates/: The User Interface**

These HTML files use the Jinja2 templating engine, which allows embedding Python-like code within HTML.

**5.3.1. index.html**

A simple welcome page.

**5.3.2. home.html**

This is the main interactive page.

* **HTML Form**: It contains a form with input fields for all the required features.
  + Categorical features like brand, model, engine\_type, etc., are implemented as dropdown menus (<select>) to prevent user error.
  + Numerical features like mileage are number inputs.
  + Binary service features are also dropdowns with "Yes" (1) and "No" (0) options.
* **Displaying Results**: It uses Jinja templating to display the prediction. The line <h2>Predicted Maintenance Cost: ₹{{ results }}</h2> will only be rendered if a results variable is passed from the Flask app (i.e., after a prediction is made).

**Chapter 6: Setup and How to Run**

**6.1. Project Dependencies (requirements.txt)**

This file lists all the Python libraries needed for the project to run. A user can install all of them with a single command: pip install -r requirements.txt. This ensures that anyone running the project has the exact same environment.

**6.2. Making the Project Installable (setup.py)**

This script uses setuptools to define the project as an installable Python package.

* **find\_packages()**: Automatically finds all packages (directories with an \_\_init\_\_.py file) in the src directory.
* **install\_requires**: Reads the requirements.txt file to define the dependencies.
* When a user runs pip install -e ., it makes the src folder available project-wide. This allows us to use imports like from src.components.data\_ingestion import DataIngestion anywhere in the project. The -e flag stands for "editable," meaning changes to the source files are immediately reflected without needing to reinstall.

**6.3. Step-by-Step Execution Guide**

To run this project on a local machine, follow these steps:

1. **Clone the Repository**:

git clone <repository\_url>

cd CAR\_MAINTENANCE\_COST

1. **Create a Virtual Environment** (Recommended):

python -m venv venv

1. **Install Dependencies**:

pip install -r requirements.txt

1. **Run the Training Pipeline** (Optional, if you want to retrain the model):

The if \_\_name\_\_=="\_\_main\_\_" block in src/components/data\_ingestion.py is set up to run the entire training pipeline.

python src/components/data\_ingestion.py

This will execute data ingestion, transformation, and model training, and save the necessary artifacts (.csv, .pkl files) in the artifacts folder.

1. **Run the Flask Application**:

python app.py

1. **Access the Web App**:  
   Open your web browser and navigate to http://127.0.0.1:5000 or http://0.0.0.0:5000. You will see the prediction form. Fill it out and click "Predict Maintenance Cost" to see the model in action.