CS702 - COMPUTING LAB

A REPORT ON THE PROJECT ENTITLED

COVERSATIONAL USED CAR PRICE PREDICTOR

Group Members:

ABHIJITH C
242CS003

ANAND M K
242CS008

I SEMESTER M-TECH CSE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA
SURATHKAL
2024 – 2025

1 Introduction

Conversational interfaces are becoming increasingly important in making human interactions with technology more natural and easy to use. These systems use Natural Language Processing (NLP) technology to understand user inputs and respond appropriately.

With the rising demand for used cars, many consumers face challenges in accurately determining the value of a vehicle, often due to a lack of information or expertise. This is where Machine Learning-based price prediction plays a vital role.

The goal of this project is to create a conversational interface that gathers key details from users to predict the price of their used car. The prediction model will use this information to estimate the car's price, which will then be communicated to the user through the interface.

2 Problem Statement and Objectives

Accurately predicting the price of a used car involves gathering and analyzing a variety of details, such as the car's brand, model, year, mileage, and other factors. Traditional methods for this, like filling out forms, can feel tedious and impersonal, leading to a poor user experience.

This project aims to address these issues by developing a Conversational Used Car Price Predictor that integrates a chatbot interface with a machine learning model. The goal is to enable users to interact naturally through conversation, rather than filling out forms, to predict the price of a used car. The chatbot will guide users step-by-step to collect necessary car details and will handle additional queries, such as explaining how the predicted price was calculated or which factors most affect the car's value.

By making the process more intuitive and engaging, this system aims to improve user experience and deliver accurate and reliable price predictions.

3 System Architecture

The system architecture for the *Conversational Used Car Price Predictor* is designed to provide an interactive, user-friendly interface for estimating used car prices. The key components of the system include the *Conversational Interface (Frontend)*, *Rasa Engine*, *Backend*, and the *Price Prediction Model*, as shown in Figure 1.

The system works as follows:

1. The user interacts with the system via a conversational interface in the frontend, providing details such as the car's make, model, year, mileage, and other relevant information.

2. These inputs are processed by the **Rasa Engine**, which handles Natural Language Processing (NLP) and conversation management.

3. The **Rasa Engine** extracts relevant parameters from the user's input and sends them to the backend via API calls.

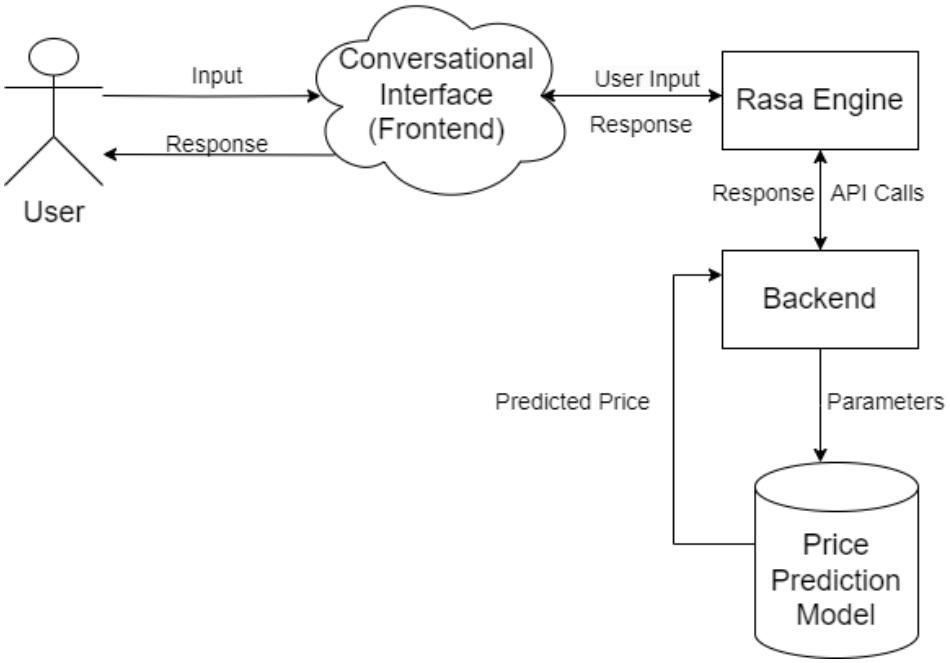

Figure 1: System Architecture of the Conversational Used Car Price Predictor

4. The backend forwards these parameters to the **Price Prediction Model**, which estimates the car's price based on the input features using a trained machine learning model.

5. SHAP (SHapley Additive exPlanations) is applied to provide an explanation of how each input feature, such as mileage or car age, contributes to the predicted price.

6. The backend returns both the predicted price and the explanation of the most significant contributing factor to the frontend.

7. The predicted price and the SHAP-based explanation are then displayed to the user through the chat interface.

This architecture ensures that the system is both interactive and explainable, allowing users to not only receive price predictions but also understand the reasoning behind them.

4 Data Preprocessing

Data preprocessing is a critical step to ensure that the machine learning model is provided with clean and well-structured data. Below are the detailed steps undertaken for preprocessing the dataset used in this project:

4.1 Dataset Acquisition

The dataset for this project was sourced from **Kaggle** in 2023, under the name `cardekho_dataset`. It contains a wide variety of features related to used cars, including attributes such as the brand, model, year, mileage, and selling price, which are essential for predicting the value of used cars.

4.2 Dropping Unnecessary Columns

Certain columns in the dataset were deemed irrelevant or redundant for our analysis. For example:

- The column `"Unnamed: 0"` was an irrelevant index and was removed.
- The `"car_name"` and `seller_type` columns, though informative, were redundant and did not contribute significantly to the model's predictive power.

4.3 Converting Vehicle Age to Year of Manufacture

To standardize the data, the `vehicle_age` column was converted to `year_of_manufacture`. This was done using the following formula:

$$\texttt{year\_of\_manufacture} = 2023 - \texttt{vehicle\_age}$$

After the conversion, the original `vehicle_age` column was dropped from the dataset.

4.4 Filtering Popular Car Models

The dataset was filtered to retain only those car models that had more than 300 entries. This filtering ensures that the machine learning model is trained on a sufficiently large number of samples for each car model. The filtering process involved:

- Calculating the frequency of each car model using `value_counts()`.
- Retaining only models with more than 300 entries to ensure robust model training.

4.5 Feature and Target Definition

For the machine learning model:

- `X` (features): Defined by dropping the columns `selling_price`.
- `y` (target): Defined as the `selling_price` of the car, which the model aims to predict.

4.6 Handling Categorical Columns

Several features in the dataset were categorical in nature, such as:

- `fuel_type`
- `transmission_type`
- `brand`
- `model`

These categorical variables needed to be converted into numerical format using encoding techniques for compatibility with machine learning algorithms. The `OneHotEncoder` was applied to these features to convert them into binary (0/1) variables.

4.7 Data Splitting

The dataset was split into training and test sets in an 80:20 ratio using the `train_test_split()` function from `scikit-learn`. This ensures that the model is trained on 80% of the data while the remaining 20% is used to evaluate the model's performance. A `random_state=42` was set to ensure reproducibility of results.

4.8 Preprocessing Pipeline

To streamline preprocessing, a pipeline was implemented using `ColumnTransformer` from `scikit-learn`. This pipeline applies `OneHotEncoder` to categorical columns while allowing other features to pass through unchanged:

```
ColumnTransformer(transformers=[('cat', OneHotEncoder(handle_unknown='ignore',
sparse_output=False), categorical_cols)], remainder='passthrough')
```

This preprocessing pipeline ensures that the categorical variables are encoded appropriately, making the data suitable for training the machine learning model.

5 Model Training Process

The model training process involves selecting an appropriate machine learning algorithm, setting up a pipeline for preprocessing and model training, tuning hyperparameters, and evaluating the model's performance through cross-validation. Below is a detailed explanation of each step involved in training the car price prediction model.

5.1 Model Selection

For this project, a **Random Forest Regressor** was chosen as the machine learning model for car price prediction. Random Forest is a powerful and flexible algorithm that handles both linear and non-linear relationships effectively. It is an ensemble learning method that constructs multiple decision trees during training and outputs the average prediction of the individual trees. This makes it robust and less prone to overfitting. Additionally, Random Forest works well with high-dimensional data and provides feature importance, making it suitable for our use case of price prediction.

5.2 Pipeline Setup

To ensure seamless and consistent data processing during both the training and testing phases, a **pipeline** was created using the `Pipeline` class from `scikit-learn`. The pipeline integrates the preprocessing steps (such as One-Hot Encoding for categorical features) and the Random Forest model into a single workflow. This approach guarantees that the same transformations applied to the training data are also applied to the test data, thereby avoiding data leakage.

5.3 Hyperparameter Tuning

To further optimize the performance of the Random Forest model, **RandomizedSearchCV** was employed for hyperparameter tuning. This method randomly samples a specified

number of hyperparameter combinations from a predefined grid, allowing for efficient exploration of the hyperparameter space. The parameter grid for the Random Forest model included the following hyperparameters:

- `n_estimators`: Number of trees in the forest, tested with values of 100, 200, and 300.
- `max_depth`: The maximum depth of each tree, with possible values of `None`, 10, 20, and 30.
- `min_samples_split`: Minimum number of samples required to split an internal node, tested with values of 2, 5, and 10.
- `min_samples_leaf`: Minimum number of samples required to be at a leaf node, with values of 1, 2, and 4.

The number of iterations for the random search was set to `n_iter=10`, meaning 10 different combinations of hyperparameters were randomly selected and evaluated.

5.4 Cross-Validation

To ensure robust evaluation of the model's performance, **5-fold cross-validation** was used during hyperparameter tuning. In cross-validation, the data is split into 5 subsets (folds), and the model is trained on 4 of these folds while the remaining fold is used as the validation set. This process is repeated 5 times, with each fold serving as the validation set once. The model's performance is then averaged over the 5 iterations to give a more reliable estimate of how it will perform on unseen data.

5.5 Model Training and Evaluation

Once the optimal hyperparameters were selected using RandomizedSearchCV, the final model was trained on the full training dataset. The model was then evaluated on the test set using the **R^2 score**, which measures the proportion of variance in the target variable (car prices) that is explained by the model. The best Random Forest model achieved an **R^2 score of 0.925** on the test set, indicating strong predictive performance.

5.6 Saving the Model

To ensure that the trained model can be used for future predictions, it was saved using the `joblib` library.

Saving the model allows for easy reuse without the need to retrain it every time a new prediction is required.

6 Identifying Influential Features

One of the most valuable aspects of the machine learning model developed for predicting used car prices is its ability to provide explanations for its predictions. This is achieved using **SHAP** (SHapley Additive exPlanations), a method that assigns contribution values to each feature for a specific prediction. SHAP values allow us to break down and understand the role of each feature in driving the predicted price, offering insights into the decision-making process of the model.

6.1 SHAP Value Calculation

Once the model is trained, SHAP values are computed for each prediction. SHAP values quantify how much each feature contributes to pushing the predicted price higher or lower than the average model output. For example, if a feature such as mileage has a positive SHAP value, it indicates that it pushes the predicted price higher; conversely, a negative SHAP value suggests that the feature lowers the predicted price.

To compute SHAP values for this project, an `Explainer` object was created using the trained `RandomForestRegressor` model. The input data, after being transformed by the preprocessing pipeline, is passed to the explainer, which returns the SHAP values for each feature. These values represent how much each feature contributed to the predicted price for that particular car.

6.2 Analyzing SHAP Values

The SHAP values are organized into a `DataFrame` for easy interpretation. This provides a clear breakdown of each feature's contribution to the predicted price. Features with positive SHAP values are identified as those that increase the predicted price, while those with negative SHAP values reduce the predicted price. The SHAP values are computed for each car, allowing for an in-depth analysis of individual predictions.

For instance, consider a test car such as the **BMW X5** (2020, Diesel, Automatic). The SHAP values provide insight into how much features like mileage, engine capacity, and fuel type contribute to the predicted price. This breakdown is valuable because it allows users to understand the key factors driving the price estimate for their car.

6.3 Identifying the Most Influential Feature

Once the SHAP values for a given prediction are calculated, the feature with the highest SHAP value is identified as the most influential factor in determining the predicted price for that specific car. This feature typically has the largest impact, either increasing or decreasing the predicted price. For example, if the `engine size` is identified as the most influential feature for a high-performance car like the **BMW X5**, it suggests that engine size is a key driver of the price for that vehicle.

6.4 Percentage Contribution Calculation

To provide a more intuitive understanding of the feature's influence, the SHAP value of the most influential feature is divided by the predicted price to calculate its **percentage contribution**. This percentage represents how much the top feature contributes to the final predicted price. For instance, if the SHAP value for `engine size` is 30% of the predicted price, it indicates that engine size is responsible for 30% of the final car price prediction.

The system outputs both the predicted price and the feature with the highest contribution, along with the percentage of its influence. This breakdown gives users a clear explanation of why the predicted price is what it is, based on the car's attributes. The transparency provided by SHAP values helps build trust in the model's predictions by offering explanations that are easy to understand.

6.5 Example Results

An example prediction for a car such as the **Hyundai i20** (2021, Petrol, Manual) would output the following results:

- **Predicted Price**: Rs.656,002.24 (approximate).
- **Most Influential Feature**: remainder__year_of_manufacture.
- **Percentage Contribution**: 24.36%.

This means that 24.36% of the predicted price can be attributed to the car's year of manufacture, highlighting the importance of this feature in the price determination process.

7 Backend API Creation

The backend of our application is built using Flask, a lightweight web framework for Python, which is designed to facilitate the creation of web applications. Flask allows us to develop RESTful APIs that serve predictions and insights derived from our machine learning model. The following outlines the structure and functionality of our API.

7.1 Endpoints

We have defined two primary endpoints for our application:

- **/predict_price**: This endpoint accepts various car attributes as query parameters and returns the predicted selling price of the car. Users can specify attributes such as brand, model, year of manufacture, kilometers driven, fuel type, transmission type, mileage, engine size, maximum power, and the number of seats.
- **/max_contribution**: This endpoint also accepts the same car attributes as query parameters and returns the highest contributing feature to the predicted price along with its percentage contribution. This information helps users understand which specific attribute has the most significant impact on the price estimation.

7.2 Input Handling

Both endpoints utilize the GET method for input handling, allowing users to retrieve predictions based on the provided query parameters. The parameters are processed as follows:

- The car attributes are retrieved from the query string of the request.
- The relevant data types are enforced; for example, integer and float values are cast appropriately to ensure the input data conforms to the expected formats.

7.3 Response Format

The API outputs predictions and contributions in JSON format. This format is widely used for data interchange and facilitates easy integration with frontend applications. JSON responses are structured to include relevant information for both predictions and contributions, allowing clients to effectively parse and utilize the data.

7.4 Implementation Details

The API is implemented in Python using the Flask framework. The core functionalities include loading a pre-trained machine learning model and using it to make predictions based on user input. Below is an overview of the main components:

```
from flask import Flask, request, jsonify
import pandas as pd
import joblib
import shap

app = Flask(__name__)

# Load the complete pipeline from Google Drive
model_path = 'best_car_price_model.pkl'
loaded_pipeline = joblib.load(model_path)
```

The model is loaded using the Joblib library, which is efficient for handling large data.

The function `predict_price` is responsible for making predictions based on the input data. It transforms the input into a DataFrame format and passes it to the model for prediction. Similarly, the function `get_max_contribution` utilizes SHAP (SHapley Additive exPlanations) to analyze the model's predictions and determine the most influential feature contributing to the predicted price.

The API endpoints are defined as follows:

```
@app.route('/predict_price', methods=['GET'])
def price_prediction():
    ...

@app.route('/max_contribution', methods=['GET'])
def max_contribution():
    ...
```

Each endpoint retrieves input parameters from the query string, processes them, and returns the corresponding predictions or contributions in JSON format.

Finally, the Flask application runs with debugging enabled, allowing for easier identification of issues during development:

```
if __name__ == '__main__':
    app.run(debug=True)
```

This setup provides a robust and flexible backend API that can be easily extended or modified to accommodate additional features and functionalities in the future.

8 Chatbot Development using Rasa

In this section, we describe the development of a rule-based chatbot using the Rasa framework. The chatbot is designed to assist users in predicting the price of a car based on its brand, model, and other attributes. The following subsections outline the key components and functionalities of the chatbot, including its configuration, intents, entities, responses, and custom actions.

8.1 Framework Overview

Rasa is an open-source framework for building conversational AI applications. It allows developers to create both rule-based and machine-learning-driven chatbots. In our case, we use Rasa's rule-based functionality to design a chatbot that can store and process car details, such as the brand and model, and provide predictions based on those details.

8.2 Domain Definition

The `domain.yml` file defines the core components of the chatbot, such as intents, entities, slots, responses, and custom actions. Below is a brief explanation of each section:

- **Intents:** These are the user's goals or actions that the chatbot can recognize. In our chatbot, the key intents include greeting the bot (`greet`), saying goodbye (`goodbye`), asking if the bot is human or a machine (`bot_challenge`), providing the car brand (`inform_brand`), and providing the car model (`inform_model`).
- **Entities:** Entities represent key pieces of information in user inputs. In this chatbot, we have two entities: `brand` and `model`, which capture the car's brand and model, respectively.
- **Slots:** Slots store information extracted from the user's input. Here, we define two slots: `brand` and `model`. Both slots are populated based on the user's input and do not influence the conversation flow directly, as indicated by the `influence_conversation: false` attribute.
- **Responses:** These are predefined messages that the bot uses to interact with the user. For example, the response `utter_greet` sends a welcoming message when the user says hello, while `utter_ask_brand` asks the user to provide the brand of the car.
- **Actions:** Custom actions are Python functions that execute additional logic during a conversation. In our chatbot, the actions `action_store_brand` and `action_store_model` store the car's brand and model in the corresponding slots.

The `domain.yml` file is structured as follows:

```
intents:
  - greet
  - goodbye
  - bot_challenge
  - inform_brand
  - inform_model
...
actions:
  - action_store_brand
  - action_store_model
```

8.3 Natural Language Understanding (NLU)

The `nlu.yml` file defines the Natural Language Understanding (NLU) component of the chatbot. It consists of training data that helps the chatbot recognize user intents and extract entities. For example, the intent `inform_brand` includes several sample sentences where the car brand is specified. The chatbot uses this data to identify when the user mentions a brand and to capture the brand as an entity.

Sample NLU data for the `inform_brand` intent is shown below:

```
- intent: inform_brand
  examples: |
    - The brand is [Maruti](brand)
    - I have a [Hyundai](brand)
    - My car brand is [Ford](brand)
    - The car is from [Toyota](brand)
```

8.4 Rules and Stories

Rasa's `rules.yml` and `stories.yml` files define how the chatbot should behave in response to specific user inputs. The `rules.yml` file contains predefined rules that map user intents to bot actions. For example, if the user says goodbye, the bot will respond with a farewell message.

An example rule for saying goodbye is as follows:

```
- rule: Say goodbye anytime the user says goodbye
  steps:
  - intent: goodbye
  - action: utter_goodbye
```

In addition to rules, the `stories.yml` file outlines conversation paths based on a series of user inputs and corresponding bot actions. Stories help the chatbot understand how to handle multi-turn conversations.

A simple story for capturing the car's brand and model is as follows:

```
- story: happy path
  steps:
  - intent: greet
  - action: utter_greet
  - action: utter_ask_brand
  - intent: inform_brand
  - action: action_store_brand
  - action: utter_ask_model
  - intent: inform_model
  - action: action_store_model
```

8.5 Custom Actions

Custom actions allow the chatbot to execute custom logic beyond predefined responses. The `actions.py` file contains Python code for handling custom actions. In our case, the

chatbot uses custom actions to store the car's brand and model in the corresponding slots.

Below is the implementation of the `action_store_brand` and `action_store_model` custom actions:

```
class ActionStoreBrand(Action):
    def name(self) -> str:
        return "action_store_brand"

    def run(self, dispatcher, tracker, domain):
        car_brand = next(tracker.get_latest_entity_values("brand"), None)
        if car_brand:
            return [SlotSet("brand", car_brand)]
        return []

class ActionStoreModel(Action):
    def name(self) -> str:
        return "action_store_model"

    def run(self, dispatcher, tracker, domain):
        car_model = next(tracker.get_latest_entity_values("model"), None)
        if car_model:
            return [SlotSet("model", car_model)]
        return []
```

These actions extract the `brand` and `model` entities from the user's input and store them in the appropriate slots, enabling the chatbot to keep track of the car details.

8.6 Conclusion

By integrating the Rasa framework, we have developed a rule-based chatbot capable of interacting with users to collect and store car details such as brand and model. The chatbot can effectively handle user inputs through predefined intents, entities, and custom actions, making it a robust tool for our car price prediction application.

9 Upcoming Work

- **Enhancing Chatbot Understanding:**
  - Add more test data to the Rasa chatbot to improve its understanding of user inputs and enhance response accuracy.
  - Expand the NLU data for handling additional user queries and responses effectively.
- **Input Validation:**
  - Implement parameter validation to ensure that the car attributes (e.g., brand, model, year) provided by users are correct and reliable for prediction.

- **General Question Handling:**
  - Enhance the chatbot's capability to handle general questions and provide informative answers to improve user engagement and satisfaction.
- **Backend Integration:**
  - Integrate the backend Flask API with Rasa to enable real-time access to the price prediction model.
- **Frontend Integration:**
  - Implement frontend integration with Rasa for seamless and interactive user experiences through web or mobile platforms.

References

[1] Rasa Technologies, "Rasa Documentation," *Rasa*, 2023. [Online]. Available: `https://rasa.com/docs/`

[2] Armin Ronacher, "Flask Documentation," *Flask*, 2023. [Online]. Available: `https://flask.palletsprojects.com/`

[3] SHAP Documentation, "SHAP (SHapley Additive exPlanations)," 2023. [Online]. Available: `https://shap.readthedocs.io/en/latest/`

[4] Leo Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, 2001, pp. 5-32. [Online]. Available: `https://link.springer.com/article/10.1023/A:1010933404324`

[5] Scikit-learn, "Random Forests," 2023. [Online]. Available: `https://scikit-learn.org/stable/modules/ensemble.html-random-forests`