Car Prediction Model API

Github repo: https://github.com/singularshift/DSPBRenderDeployment
Render API: https://dspbrenderdeployment.onrender.com/docs

1. Introduction: context of the project

The objective of this project is to apply machine learning techniques to predict the price of used cars, and make it available through an API. The dataset used for this project is available on Kaggle. The goal is to develop a robust model capable of estimating car prices based on a set of features, leveraging data preprocessing, feature engineering, and advanced regression techniques to achieve high accuracy, and create a FastAPI API with our model and deploy it on Render. In this report, we describe our methodology, from data preprocessing to model comparison to the deployment of our API.

2. Data cleaning and preprocessing

The dataset contained a mix of categorical and numerical features that required extensive preprocessing to ensure consistency and usability for machine learning models. The data cleaning and transformation steps were designed to standardize column names, encode categorical variables, clean numerical features, and remove outliers for better model performance. In the following, we'll go through our workflow in depth to give you a better reasonable understanding about our choices.

Column name standardization

To maintain uniformity and ease of access when handling the dataset programmatically, column names were standardized. All names were converted to lowercase, spaces were replaced with underscores, and unnecessary characters such as periods were removed. This step ensured consistency across the entire pipeline and prevented syntax-related errors when working with column names.

Feature categorization

Before applying transformations, features were categorized into two main groups:

Continuous features: `prod_year`, `engine_volume`, `mileage`, `airbags`

Categorical features: `manufacturer`, `model`, `category`, `leather_interior`, `fuel_type`, `gear_box_type`, `drive_wheels`, `doors`, `wheel`, `color`

This categorization was crucial in determining the appropriate preprocessing methods for each type of feature.

Preprocessing of specific Columns

Categorical encoding

Categorical variables were converted into numerical representations using categorical encoding. Since these variables contained textual values, they were first converted to lowercase for consistency and then encoded into numeric categories using `category` data type conversion. The following features were transformed using this method: `manufacturer`, `model`, `category`, `leather_interior`, `fuel_type`, `gear_box_type`, `drive_wheels`, `doors`, `wheel`, and `color`

This transformation allowed the model to interpret categorical data numerically without introducing unnecessary complexity.

Processing engine volume and turbo feature creation

The `engine_volume` column contained both numerical values and text labels indicating the presence of a turbo engine. To extract this information, a new binary feature, `turbo`, was created by checking whether the `engine_volume` column contained the word "Turbo" and assigning `1` if present, otherwise `0`. Also, the text "Turbo" was removed from `engine_volume`, and the remaining numerical values were converted into float format. This step ensured that turbocharged engines were correctly accounted for as a separate feature while preserving accurate engine volume measurements.

Mileage cleaning

The `mileage` column contained text artifacts such as "km" at the end of values. These were removed, and the remaining values were converted to numerical format for consistency and ease of use in model training.

Numeric conversions

Some columns, such as `cylinders` and `airbags`, were originally stored in non-numeric formats. These were explicitly converted to integer data types to maintain numerical consistency.

Outlier removal

Outlier removal was a crucial step to ensure that extreme values did not disproportionately impact the model's predictions.

Mileage outlier removal using the IQR method

To detect and remove extreme values in `mileage`, the **Interquartile Range (IQR) method** was used:

The first quartile (Q1) and third quartile (Q3) were calculated.

The interquartile range (IQR) was determined as `IQR = Q3 - Q1`.

The lower bound was set as `Q1 - 1.5 * IQR`, and the upper bound as `Q3 + 1.5 * IQR`.

Any values falling outside this range were removed.

This approach effectively removed extreme mileage values while preserving a majority of the dataset.

High price outlier removal

Since an exceptionally high price outlier could significantly skew model performance, the single highest-priced vehicle in the dataset was identified and removed. This step prevented the model from being overly influenced by extreme, unrealistic price values. We also assume that there might have been some data errors, due to unrealistic high prices/mileages, that were excluded this way.

Feature dropping

To enhance model efficiency, unnecessary features were removed, the `levy` column was dropped since it contained a high proportion of missing values and did not provide meaningful information for price prediction and the `id` column was removed before model training, as it was merely an identifier with no predictive relevance.

Final preprocessing outcomes

After completing these preprocessing steps, the dataset was transformed into a structured, standardized format suitable for training machine learning models. The key benefits of these transformations included that we now have consistent column naming conventions for easier handling and reproducibility, removal of text artifacts to ensure numerical accuracy, categorical feature encoding to allow machine learning models to interpret textual data effectively, outlier removal to enhance prediction stability and accuracy, and creation of new features (e.g., turbo indicator) to improve model interpretability and performance.

With the data now preprocessed, the next step involved training and evaluating different machine learning models to determine the most accurate approach for price prediction.

3. Model selection and performance evaluation

Several regression models were trained and evaluated based on their Mean Squared Error, which measures the average squared difference between the actual and predicted car prices. The models tested included:

Linear Regression: A basic model that assumes a linear relationship between input features and output price. It performed poorly due to the complex non-linear relationships in the dataset.

Lasso Regression: Similar to linear regression but with L1 regularization, which helps with feature selection by reducing some coefficients to zero. It also did not perform well.

Ridge Regression: Uses L2 regularization to prevent overfitting, but it was unable to capture the non-linearity in the data.

Decision Tree Regressor: A tree-based model that learns non-linear patterns in the data. It provided better results than linear models but tended to overfit the training data.

Random Forest Regressor: An ensemble method that combines multiple decision trees to improve accuracy. It performed better than individual decision trees but was still outperformed by boosting algorithms.

Gradient Boosting Regressor: An ensemble method that builds trees sequentially, correcting previous errors. It showed significant improvement over previous models.

XGBoost Regressor: An optimized version of gradient boosting that is computationally more efficient and includes regularization.

Model performance comparison

The results from initial training showed that ensemble methods, particularly Gradient Boosting and XGBoost, significantly outperformed traditional linear regression techniques. The MSE values for the best-performing models were:

Model	MSE
Linear Regression	435,762,200
Lasso Regression	435,762,400
Ridge Regression	435,762,500
Decision Tree Regressor	601,781,000
Random Forest Regressor	302,616,000
Gradient Boosting Regressor	8,355,662
XGBoost Regressor	10,189,000

Since Gradient Boosting achieved the lowest MSE, it was chosen as the final model.

The superior performance of Gradient Boosting can be attributed to its ability to sequentially learn from errors, focusing on difficult-to-predict instances. Unlike traditional models, boosting methods iteratively refine their predictions, leading to greater accuracy. Another reason why Gradient Boosting

outperformed XGBoost in this case could be due to the specific nature of the dataset. XGBoost, while generally more efficient, requires careful tuning of additional hyperparameters like `colsample_bytree` and `gamma`, which might not have been fully optimized in this instance. Overall, Gradient Boosting was selected for deployment due to its balance between accuracy and interpretability, making it a suitable choice for predicting used car prices with high precision. To optimize the predictive power of the Gradient Boosting model, we selected six key features: `prod_year`, `engine_volume`, `mileage`, `cylinders`, `airbags`, and `turbo`. This selection process was guided by multiple factors, ensuring that the model maintained high accuracy while avoiding unnecessary complexity.

Feature Importance analysis

A Random Forest Regressor was initially used to evaluate feature importance, identifying the attributes that had the strongest impact on predicting car prices. The six selected features consistently showed high predictive value, making them the most relevant contributors to price estimation. By limiting the feature set to only these highly significant variables, the risk of overfitting was minimized while retaining strong model performance.

Feature types and business logic

The chosen features were a combination of numerical and binary indicators that captured key aspects of a car's valuation:

Numerical features (5): `prod_year`, `engine_volume`, `mileage`, `cylinders`, `airbags`

Binary feature (1): `turbo` (derived from `engine_volume` to indicate turbocharged engines)

Each feature was included based on its real-world relevance:

`prod_year`: Directly correlates with vehicle depreciation and aging effects.

`engine_volume` & `turbo`: Represent engine capacity and performance, impacting both fuel efficiency and power.

`mileage`: A primary indicator of vehicle wear and tear, strongly affecting resale value.

`cylinders`: Tied to engine power and fuel consumption, often influencing price.

`airbags`: A crucial safety feature that adds value, particularly in newer car models.

From an API usability standpoint, these six features were chosen for their practical availability and reliability in real-world applications as they are easily obtainable from car listings or user-provided input, objective measurements with minimal ambiguity, standard across different car models and manufacturers, and reliable indicators of car value that users can provide with confidence.

4. Hyperparameter tuning: finding the best model configuration

Once Gradient Boosting and the features were selected, hyperparameter tuning was performed using Grid Search. The following parameters were optimized:

n_estimators: Number of boosting iterations (values tested: 100, 300, 500)

learning_rate: Controls the step size at each iteration (values tested: 0.01, 0.1, 0.2)

max_depth: Maximum depth of the trees (values tested: 3, 5, 7)

subsample: Fraction of samples used to fit base learners (values tested: 0.6, 0.8, 1.0)

The best configuration found was:

{'learning_rate': 0.01, 'max_depth': 7, 'n_estimators': 100, 'subsample': 0.6}

This configuration achieved an MSE of 8,355,662, significantly improving prediction accuracy.

5. FastAPI deployment, API functionality and Streamlit UI

After selecting and tuning the best-performing model, it was necessary to make it accessible for real-world use. This was achieved by deploying the model using FastAPI, a high-performance web framework for building APIs. The key goal was to allow users to send car feature data and receive price predictions in real time. The responding code was written in app_gbr.py and loaded the saved model from our pickle file created earlier. The API allows users to obtain car price predictions by sending a POST request to the `/predict` endpoint with car feature values in JSON format. Upon receiving the request, the input data is processed using the same preprocessing pipeline applied during model training, ensuring consistency through one-hot encoding for categorical variables and standardization for numerical features. The cleaned and transformed data is then passed into the trained Gradient Boosting model, which generates a price estimate based on the provided attributes. Finally, the predicted price is returned in JSON format, making it easy for users to integrate the API's output into other applications or workflows. For a smoother user experience, we implement a simple Streamlit UI to expose the API.

Example API request and response

Request: POST /predict

{ "turbo": 1, "airbags": 6, "prod_year": 2018, "cylinders": 4, "engine_volume": 2.0, "mileage": 50000 }

Response: { "predicted_price": 12500.75 }

To ensure seamless model deployment, the encoder and scaler objects used during training were saved alongside the trained model. This ensures that the API processes new input data in the exact same way as during training, preventing discrepancies between training and inference.

6. Conclusion and future work

This project successfully built, optimized, and deployed a machine learning model for used car price prediction. By leveraging Gradient Boosting, the model achieved high accuracy, and its performance was further enhanced through hyperparameter tuning. The integration of FastAPI enabled real-time price estimation, making the system practical for real-world applications.

One of the most crucial aspects of this project was the extensive data preprocessing, which significantly improved model performance by ensuring that both categorical and numerical variables were properly encoded and scaled. Without these preprocessing steps, inconsistencies in the dataset could have negatively impacted the accuracy and generalizability of the model. The evaluation of multiple machine learning models revealed that ensemble methods such as Gradient Boosting and XGBoost outperformed traditional models, highlighting the power of tree-based learning in handling complex regression tasks. These models were able to capture intricate relationships between car attributes and price more effectively than linear regression-based approaches. Further improvements were achieved through hyperparameter tuning using Grid Search, which led to a substantial reduction in prediction error. By optimizing key model parameters such as learning rate, maximum tree depth, and the number of estimators, the Gradient Boosting model achieved a well-balanced trade-off between accuracy and computational efficiency. Finally, deploying the trained model via FastAPI provided a scalable and user-friendly solution, allowing users to interact with the prediction system in real-time. The API architecture ensured seamless integration into various applications, making the price estimation model accessible for practical use in business and consumer applications alike.

While the model performed well, several enhancements could be explored. The first one would be Incorporating additional features, such as historical pricing trends, brand reputation, or car maintenance history, exploring deep learning models to capture even more complex relationships in the data, enhancing interpretability using tools like SHAP (SHapley Additive Explanations) to understand how each feature influences the price prediction, and expanding API functionalities, such as adding endpoints for batch predictions or visualizing price trends.