

Milestone V Report - Model and Utility Application Implementation

Predicting Used Car Prices using Machine Learning

CSCE 3602 - Fundamentals of Machine Learning

Omar Bahgat
Computer Engineering Department
The American University in Cairo
Cairo, Egypt
omar_bahgat@aucegypt.edu

Omar Saleh
Computer Engineering Department
The American University in Cairo
Cairo, Egypt
Omar_Anwar@aucegypt.edu

Introduction

The final stage of the project involved developing the model as per the design outlined in the preceding phase. Initially, we experimented with various XGBoost parameters to identify the most optimal ones. Once the optimal parameters were determined, we experimented with a custom loss function to boost our R-squared score and proceeded to train our model using our *cleaned_dataset*. We also focused on implementing the utility application, which comprises four primary components: the user interface, user input preprocessing, price prediction, and online learning. This report will delve into the specifics of the processes undertaken during this phase in comprehensive detail.

Part I: Model Implementation

Having identified XGBoost as the most promising model from the previous phase, this phase we delved into optimizing its performance through two key strategies: a custom loss function and hyperparameter tuning.

Custom Loss Function

Our custom loss function is a hybrid of Mean Squared Error (MSE) and Linear Loss and is based on a threshold. It's designed to take advantage of the strengths of both these loss functions, hence the term "hybrid".

Mean Squared Error (MSE): When the absolute error (the absolute difference between the predicted and actual values) is less than the threshold, the function calculates the gradient and Hessian as if it were using a Mean Squared Error loss function. The gradient is $3 * err^{**2}$ and the Hessian (square matrix of the second-order partial derivatives) is $6 * err$. This part of the function emphasizes smaller errors and encourages the model to pay more attention to reducing these errors.

Linear Loss: When the absolute error is greater than or equal to the threshold, the function calculates the gradient and Hessian as if it were using a Linear Loss function. The gradient is $2 * err$ and the Hessian is 2. This part of the function ensures that larger errors are not overly penalized, making the model more robust to outliers.

After tuning we have found the optimal threshold for our model to be 0.08.

Why this hybrid approach is beneficial

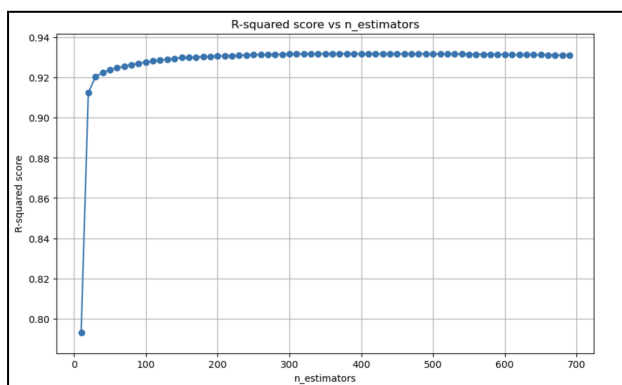
In the context of predicting used car prices, the hybrid loss function offers a unique advantage by effectively handling a wide range of price variations. The function's quadratic component focuses on minimizing small errors, which is crucial for accurately predicting prices of lower-end cars where small price differences can significantly impact buyer decisions. Conversely, the linear component ensures the model's robustness to outliers, accommodating the high variability in prices of premium or

unique cars. The adjustable threshold provides the flexibility to fine-tune the model's sensitivity to different error sizes, optimizing its performance. In Summary, the hybrid loss function provides a balance between sensitivity to small errors and robustness to outliers, making it well-suited for the diverse range of errors that can occur in used car price prediction.

Hyperparameter Tuning

XGBoost boasts a rich set of hyperparameters, each significantly impacting its behavior. We employed a grid search approach, meticulously exploring a range of values for critical parameters:

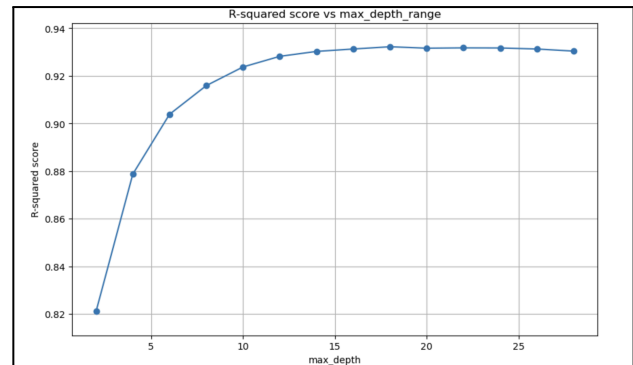
n_estimators (Number of Trees): This parameter dictates the number of decision trees incorporated into the final ensemble model. We systematically evaluated various values to pinpoint the optimal number that strikes a balance between model complexity and its ability to generalize to unseen data. A higher number of trees can enhance model accuracy, but excessive trees can lead to overfitting, where the model memorizes the training data and performs poorly on new data.



We found the best score at 370 trees.

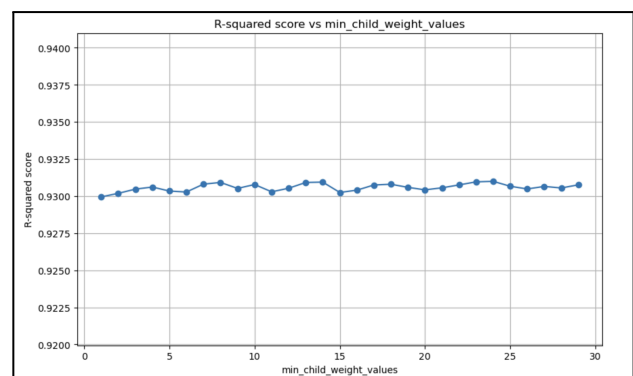
max_depth (Maximum Depth of Trees): This parameter exerts control over how intricate each individual tree is. Tuning this value helps mitigate overfitting. Shallower trees (lower

depth) are generally easier to understand but might not capture complex relationships within the data. Conversely, deeper trees can model intricate patterns but are more susceptible to overfitting.



We found the best score at depth 22.

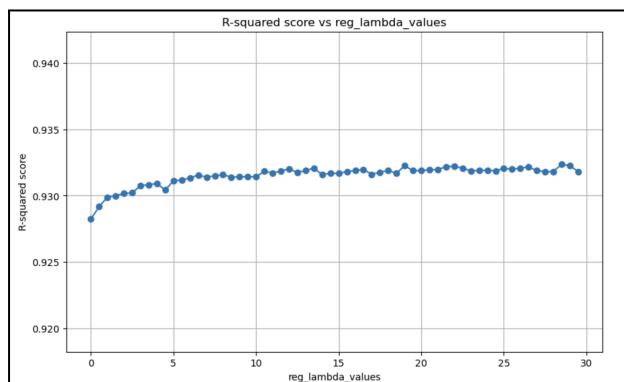
min_child_weight (Minimum Weight for Child Nodes): This parameter influences the minimum number of data points required in a node before it's further split. Adjusting this value helps control model complexity and overfitting. A higher weight necessitates more data points for a split, leading to a simpler model. Conversely, a lower weight might result in a more complex model that overfits the data.



We found the best score at weight 14.

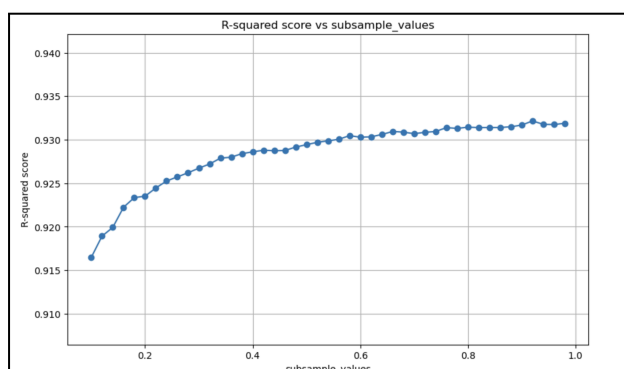
reg_lambda (L2 Regularization Weight): This parameter introduces a penalty term during the training process, discouraging overly complex models. This helps deter overfitting by penalizing models with a high number of weights. By adjusting the reg_lambda value,

we can control the trade-off between model complexity and its ability to fit the training data.



We found the best score at $\text{reg_lamda} = 14.6$

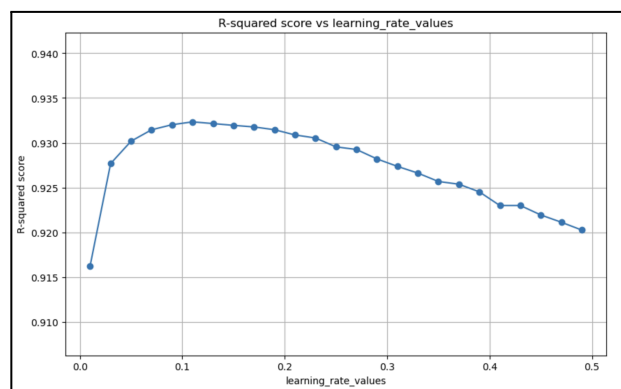
Subsample Size (subsample): This hyperparameter controls the proportion of training data used to build each individual tree in the XGBoost ensemble. A value of 1 uses the entire dataset for each tree, while values less than 1 (typically between 0.5 and 1) perform sampling with replacement for each tree. This technique, called bagging, helps prevent overfitting by introducing diversity into the ensemble. By using a different subset of data for each tree, the model is less likely to memorize specific patterns in the training data and can generalize better to unseen data.



We found the best score at 0.92 subsample size

Learning Rate : This hyperparameter controls the step size taken by the model when updating its weights during each iteration of the

training process. A higher learning rate leads to larger steps, potentially enabling faster convergence but also increasing the risk of overshooting the optimal solution. Conversely, a lower learning rate takes smaller steps, leading to slower convergence or even no convergence but potentially achieving a more accurate minimum. Finding the optimal learning rate is crucial for balancing the speed and accuracy of the training process.



We found the best score at 0.13 for the learning rate

Through this meticulous grid search, we identified the optimal combination of hyperparameters that yielded the best possible performance on the validation set. This fine-tuning resulted in a great R-squared score of 0.9323.

Part 2: User-input Preprocessing

Since the user of our utility application cannot provide preprocessed data directly, we had to have dedicated components to transform the user-entered data to a format ready for prediction and retraining by our model.

Files needed for preprocessing

- **'features.json'**: This file stores the column names of the dataframe used for the initial training of the model. It serves as a reference for the structure of the input data

and will be used later for user-input processing.

- **'stats.json'**: This file stores statistical information required for preprocessing numerical features. It stores key-value pairs `{'feature' : {mean, std}}`, encapsulating the mean and standard deviation values for each feature. This enables efficient preprocessing of numerical features without the need to recalculate statistics for the entire dataset. The `stats.json` file is continuously updated after each retraining iteration.

Initialization

A `'car_info'` dataframe is created with dimensions (1, number of features), initialized with column names retrieved from the `'features.json'` file. This dataframe is initially empty, with all values set to zero.

Handling Categorical Features

To transform categorical features entered by the user, we begin by concatenating the feature name with the user-entered value. This concatenation generates a string that aligns with the column names in the previously created `'car_info'` dataframe. Subsequently, we check if this concatenated column name exists in the dataframe. If found, we assign a value of 1 to that column to indicate the presence of the corresponding category. In cases where the concatenated column name is not found, we assign the value to the column named by concatenating the feature name with "Other." This approach ensures that all categorical features are appropriately represented, even if the specific category entered by the user is not explicitly defined in the original dataset.

As an example, let's consider the `'model'` feature. If the user enters `'Corolla'` as the model name, we concatenate `'model_'` with `'Corolla'` to create the string `'model_Corolla'`. If this

column name exists in the `car_info` dataframe, we assign `car_info.loc[0, 'model_' + 'Corolla'] = 1`. However, if the column name is not found, we assign `car_info.loc[0, 'model_' + 'Other'] = 1`.

Handling Numerical Features

Numerical features entered by the user are Z-scaled using the mean and standard deviation values stored in the `stats.json` file. This ensures consistency with the scaling applied during the model training phase. For each numerical feature entered by the user, its value is scaled accordingly.

Part 3: Utility Application

Client Application GUI

Framework

The graphical user interface (GUI) of the client application was developed using the Tkinter library in Python. Tkinter was selected as the primary framework due to its user-friendliness and high integration capabilities with other Python libraries

Data input fields

The GUI includes a comprehensive set of features designed to improve the user experience and enhance functionality. To begin with, users are presented with data input fields corresponding to twenty distinct features -

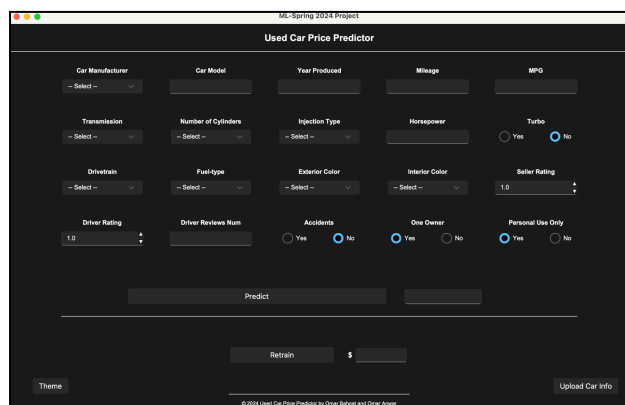
`['manufacturer', 'model', 'year', 'mileage', 'mpg', 'transmission', 'cylinders', 'injection_type', 'horsepower', 'turbo', 'drivetrain', 'fuel_type', 'exterior_color', 'interior_color', 'seller_rating', 'driver_rating', 'driver_reviews_num', 'accidents', 'one_owner', 'personal_use']` -

required for accurate car price prediction. Each input field is accompanied by a clear label, providing users with guidance on the type of information to be entered. In addition to standard input fields, the GUI incorporates dropdown menus for certain features where users are presented with predefined choices.

This simplifies data entry by offering users a selection of options to choose from, eliminating the need for manual input and minimizing potential errors. Furthermore, for features with discrete choices, radio buttons are provided, allowing users to make selections with a single click.

Buttons

At the heart of the application lie two key functionalities: the "Predict" and "Retrain" buttons. The "Predict" button requests a price prediction from the server, delivering real-time estimates based on user inputs. Conversely, the "Retrain" button facilitates continuous refinement of the underlying machine learning model by taking the actual car price from the user and retraining the model. Another feature implemented is the "Theme" button which enables users to toggle between dark and light modes.

The image shows a web application titled "Used Car Price Predictor" with a dark theme. It features a grid of input fields for car details: Car Manufacturer, Car Model, Year Produced, Mileage, and MPG. Below these are Transmission, Number of Cylinders, Injection Type, Horsepower, and Turbo (Yes/No). Further down are Drivetrain, Fuel Type, Exterior Color, Interior Color, and Seller Rating. At the bottom of the input section are Driver Rating, Driver Reviews Num, Accidents (Yes/No), One Owner (Yes/No), and Personal Use Only (Yes/No). A "Predict" button is centered below the inputs. Below the prediction area is a "Retrain" button followed by a price input field starting with a dollar sign. At the very bottom, there is a "Theme" button on the left and an "Upload Car Info" button on the right. A small copyright notice is visible at the bottom center.

Graphical User Interface

Client Application Flow

Upon opening the client application, users are prompted to input details about the car they wish to know its price. Once all required information is entered, users proceed by clicking the "Predict" button to initiate the price prediction process.

Before sending the data to the server for processing, the application performs validation

checks on all input fields to ensure that they contain valid data. The validation mechanism includes checks for the presence of data in all input fields and employs custom validation functions for numerical fields. These custom functions prevent users from inputting invalid data such as strings, negative numbers, numbers starting with zero, or values exceeding predefined ranges for each feature.

If the validation process succeeds, the client application constructs a Python list containing the inputted car details. This list is then sent via an API request to the server. Upon receiving the request, the server predicts the price and returns it to the client application. The client application displays the predicted price in the designated textbox.

Server Application Flow

The server-side operation begins with the extraction of data from the `'stats.json'` and `'features.json'` files, storing their contents into Python dictionaries. These dictionaries are crucial for the preprocessing of user-input features which was described above.

Following this, the server conducts preprocessing on the input data, transforming it into a dataframe suitable for our custom XGBoost model which is already trained on our data and saved in a binary format. Once the data preprocessing is complete, the dataframe is passed to the prediction function. Within this function, the preprocessed data is converted into a DMatrix - an internal data structure utilized by XGBoost, optimized for memory efficiency and training speed. Afterwards, the trained XGBoost model is loaded into memory using the `model.load_model('file_name')` function and the model predicts the car price. The server then returns the predicted price back to the client application.


The server utilizes two main libraries for handling data and facilitating communication with the client. The json library manages JSON data, extracting information from files like stats.json and features.json while the http.server library facilitates smooth data exchange between the client and server components.

Online Learning

Upon pressing the retrain button, the application verifies that all input fields are filled, employing the same validation mechanism done for predicting. Additionally, it verifies whether the user has inputted an actual car price which is necessary for re-training. After successful validation, input features are compiled into a Python list and are sent to the server via an API, where they undergo preprocessing and are appended to a *samples* list.

Retraining occurs in intervals of three samples. Once the *samples* list contains three dataframes which represent three cars' features, the server loads the model, converts the *samples* list into a DMatrix, and trains the model using this DMatrix. The updated model is then saved for future use. This systematic approach ensures the model's continual improvement by incorporating new data.

Application Demo Link

 [ML-Spring 2024 Project Demo](#)