Faculty of
Mathematics
and Informatics

VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
DATA SCIENCE
MASTER'S STUDY PROGRAMME

A MODEL FOR DETERMINING THE MARKET PRICE OF A CAR

Master's thesis

Author: Vytautas Jaras
VU email address: vytautas.jaras@mif.stud.vu.lt
Supervisor: (Dr. Tomas Plankis)

Vilnius

2024

Santrauka

Šiame magistro rašto darbe įvertinamas įvairių regresijos metodų gebėjimas prognozuoti automobilių rinkos kainas. Darbe aprašomi ir analizuojami tokie modeliai kaip OLS regresija, Atraminių Vektorių regresija, K-Artimiausių Kaimynų regresija, Sprendimų Medžių regresija, Gradiento Augimo regresija bei MLP regresija. Tyrimas susideda iš duomenų paruošimo, modeliavimo ir parametrų derinimo etapų, taip pat yra pritaikoma modeliavimo sistema, skirta prognozuoti kiekvienos automobilių markės kainas atskirai. Skirtingai nuo kitų panašių tyrimų, šiame darbe kartu su įprastomis automobilių savybėmis įtraukiami mėnesiniai fiktyvūs kintamieji. Tyrimas parodė, kad efektyviausias metodas yra Sprendimų medžių regresija, pasižyminti stipriais rezultatais bei greitu veikimu. Taip pat, arklio galia ir automobilio amžius buvo įvardinti kaip reikšmingiausi kintamieji. Modelių rezultatai galėtų būti dar tikslesni, įtraukiant daugiau nepriklausomų kintamųjų, pavyzdžiui, automobilio komplektacijos lygį ar automobilio eismo įvykių istoriją, taip pat naudojant kokybiškesnius duomenis su mažesniu trūkstamų duomenų kiekiu. Ilgesnio laikotarpio laiko eilučių duomenys kartu su makroekonominiais rodikliais, tokiais kaip infliacija, taip pat padėtų geriau paaiškinti automobilių kainų svyravimus.

Raktiniai žodžiai: OLS regresija, Atraminių Vektorių regresija, K-Artimiausių Kaimynų regresija, Sprendimų Medžių regresija, Gradiento Augimo regresija, MLP regresija, Automobilio kaina.

Abstract

This master thesis evaluates the ability of various regression methods to predict car market price. It describes and assesses different models such as OLS Regression, Support Vector Regression, K-Neighbors Regression, Decision Trees Regression, Gradient Boosting Regression, and MLP Regression. The study outlines the steps for data preparation, modeling and hyperparameter tuning, while also proposing a modeling framework for predicting prices for each car brand separately. Moreover, distinct from similar studies, this research includes monthly dummy variables along with the more conventional car features. The study finds that Decision Trees Regression emerges as the most effective method, demonstrating high performance while also being time-efficient. In addition, Horse Power and Age at Sale stand out as the features having the most importance. However, the results could be further improved by including more independent variables such as trim level or accident history and using higher quality data with less missing information. Moreover, integrating extended time series data with macroeconomic indicators could also provide a clearer picture of the variations in car prices.

Keywords: Cross-sectional data, OLS Regression, Support Vector Regression, K-Neighbors Regression, Decision Trees Regression, Gradient Boosting Regression, MLP Regression, Car price.

Contents

Notation

- $\sum$: Denotes a sum over a set of terms.
- $|| \cdot ||_2$: Represents the L2 norm or Euclidean norm, calculating the square root of the sum of squared elements of a vector.
- $\min$ / $\max$: Denote the minimum or maximum value, typically of a function or set.
- $\mathbf{X}$, $\mathbf{y}$: Bold symbols represent matrices or vectors.
- $\hat{\cdot}$: Typically used to denote an estimated or predicted value.
- ∇: Represents the gradient of a function.
- α, β: Commonly used to represent parameters or coefficients in statistical models.
- ^{T}: Transpose operation on matrices or vectors.
- **arg min**: The operator $\arg\min_x f(x)$ identifies the value of x at which the function $f(x)$ attains its minimum value. It is widely used in optimization to determine the point at which a function is minimized.

Introduction

In recent years, the new and used car market has been a widely discussed topic because of a surge in prices. According to Kelley Blue Book, car valuation and automotive research company, the average price of a new car in the United States, as of September 2023, was $48,000 - $6,000 more than during the same month in 2021 and $10,000 more than in September 2020 [1]. Such rise in new car prices could be attributed to multiple factors which were present during the COVID-19 pandemic. First of all, driven by generous fiscal stimulus, consumers opted for bigger vehicles - SUVs being the most popular type of vehicle in the U.S. - one of the biggest new car markets in the world. The demand for new vehicles kept rising, however, the supply side had a hard time keeping up. COVID-19 had a large impact on disrupting workplace operations. Furthermore, the automotive industry supply shortages were even more amplified by a world-wide semiconductor shortage, since electronics are an important part of a modern vehicle. Another important point to mention is that car production has been on a downward trend since the mid-1990s, which was only amplified during the COVID-19 period [6]. Naturally, a high demand for new vehicles and car manufacturers inability to meet customer needs led to low dealer inventories and increase in prices. Consumers were forced either wait for availability or pay well above the Manufacturer's Suggested Retail Price (MSRP).

The increase in prices and lack of supply in the new car market have also trickled down to the used car market. Depleted car dealership inventories meant more people started looking at used cars as an alternative. Furthermore, a strong online infrastructure greatly benefited the used car market. According to Fox Dealer, a company which specializes in online car sales software, there has been 40% increase in online used car activity, since the beginning of the pandemic up until March 2021. However, even though the first interaction with a customer is made online, customers still show a preference for in-person meetings and car inspections before making a purchase. Transparency remains a key concern for used car shoppers and should be continuously improved in the development of online marketplaces [3].

Furthermore, according to McKinsey & Company's article, data and analytics are poised to transform the used car market, simultaneously opening up new opportunities for growth. The report projects that the institutional retail used car market in both Europe and the United States could generate $1.2 trillion in revenue, only in 2023, with market having a lot of space for new innovation. Consumer purchasing habits are increasingly shifting online, which should guide used car companies to storing and utilizing market data [8].

A robust and efficient price prediction method could be useful for buyers in search of a better deal and for the transparency and efficiency of the market itself. A price estimate for a specified type of vehicle, made visible on an online marketplace, could provide an incentive for private sellers or dealerships to create listings with prices closer to the fair market value. In addition, training a model on car dealer data, could provide a valuable information that would help dealers to predict their profit margins. Furthermore, such model can be trained on both new and used car data, accommodating for the variability in new car prices, which are often determined by the dealership rather than being fully set by the manufacturer.

Goal of the Study

The study's goal is to evaluate and compare multiple regression methods for predicting car market prices and to suggest the optimal method. This method must not only be theoretically sound and robust but also practical for real-world application, being both reasonably fast and easy to maintain.

Objectives of the Study

1. Conduct a thorough literature review on price prediction methods and their applications in the automotive market.
2. Describe the methods that were utilized during the study.
3. Present the process of preparing the CIS automotive data set for training and evaluating the models.
4. Implement methodology for predictive model development, including data pre-processing and hyper-parameter optimization.
5. Tailor and optimize models for predicting prices for each car brand, reflecting brand-specific market dynamics.
6. Evaluate the performance of each model, identify the most effective ones, provide insights for practical application and draw a conclusion

1 Literature Review

This section presents a review of the literature which helped to shape this study. It is organized to reflect the areas which were the main building blocks of the research process, guiding the creation of data processing and modeling framework.

1.1 Data Selection

Even though the main focus of research is often the statistical or machine learning methods, selecting the right data for training and testing these methods is equally, if not more, important. In research focusing on car price prediction, typically two types of data are used: the first type is data scraped from car advertisement websites, and the second is data from car sales records.

For instance, Shanti et al. [27] have scraped car adverts on multiple websites spanning over 4 years. Nonetheless, data scraping can become a time-consuming task when a larger data set is required, particularly with data spanning over extended periods of time. When saving time, sometimes, one snapshot at one point of time could be taken. For example, Longani et al. [13] used 4057 records of car adverts scraped on the same date. However, this approach makes it challenging to capture the variability in prices that occurs over time.

In addition, car adverts might not be the best source of information. One could argue that car advert asking price does not completely reflect the actual market value of a particular vehicle. A seller can ask for an unrealistic price, also without any buyer willing to purchase the vehicle unless maybe in the future price will get dropped. Moreover, buyers can try to negotiate the price over the phone or in person before making the final purchase. This ambiguity about the fair market value of a particular vehicle could be solved if a researcher used data from car sales records. For example, a study by Lessmann et al. [11] uses 450,000 observations which refer to actual sales of six different car models in the used car market. Such data set was provided by an anonymous leading German car manufacturer. This is an example of an advantage that businesses training a model on an in-house historical data could have over independent researchers. One could argue that such data is ideal for training a model to predict car prices, as it comprises actual sales records, encompasses a vast array of data across different car models, and reflects the variation of car prices over time.

1.2 Data Pre-Processing

After selecting the data set, it is crucial to properly prepare it according to the type of problem that is being addressed and the statistical methods chosen to solve it. Lessmann et al. [11] addresses the possible non-linear relationship between used car prices. Main reasons that would explain such behavior could be car depreciation happening faster in the early years of car life cycle [5], announcements of new models negatively affecting resale prices of older models [14] and consumers suffering from various cognitive biases [9]. For example, linear regression might require logarithmic transformation of a response or explanatory variables [7]. However, newer algorithms such as Neural Networks take non-linearity into account [11].

Moreover, it is important to pay attention to the optimization technique that is used by the methods. For instance, Artificial Neural Networks and Linear Regression can utilize gradient descent optimization,

where feature scaling can enhance the performance of these methods. As noted by Shanti et al. [27], feature scaling improves the convergence speed of gradient descent. In addition, the performance of methods that rely on distance as a measure of similarity, such as K-Nearest Neighbors, can also benefit from scaling, as it eliminates the bias caused by varying scales of different feature values.

1.3 Modeling Techniques

In order to predict prices of used cars, one can employ various regression methods fitted on available features. However, some methods are able to deal with this task better than others. A study by Lessmann et al. [11] evaluates wide range of modeling techniques available in MATLAB. The individual methods used in this study include Multivariate Linear Regression, Stepwise Linear Regression, Ridge Regression, Lasso Regression, Multivariate Adaptive Regression Splines, Artificial Neural Networks, Support Vector Regression, Regression Trees, K-Nearest Neighbors. The authors also consider homogeneous ensemble methods such as Bagged Regression Trees, Bagged ANN, Boosted Regression Trees, and Random Regression Forest, as well as heterogeneous ensemble methods—including Ensemble Selection, Simple Average, Trimmed Average, Weighted Average, Stacking, and Simple Average over the best N models. Moreover, the study explores various combinations of parameters and response variable transformations during model training. Previously listed models were trained and tested on the data segmented into 6 splits - each corresponding to a different car model. Linear individual prediction models such as Multivariate Linear Regression, Lasso Regression, Stepwise Linear Regression and Ridge Regression were among the least accurate methods based on their corresponding Mean Absolute Error (MAE) scores. These methods were not confined solely to linear relationships, as logarithmic transformations of the response variable were also applied. Nevertheless, methods that automatically handle non-linear relationships demonstrated significantly better results. The top eight methods in terms of performance are all ensemble methods, with only the least effective among them, Boost, falling behind the two superior single prediction methods, Artificial Neural Network (ANN) and Support Vector Regression (SVR).

Another, more recent, study on used car price prediction by Valarmathi et al. [28] compares three machine learning techniques: Support Vector Regression, Random Forest Regression, and CatBoost Regression. The methods were evaluated using R-squared, Mean Square Error (MSE), Root Mean Square Error (RMSE), Mean Absolute Error (MAE), and Accuracy. Random Forest Regression emerges as the best one across all the metrics, with the Accuracy score of 87.19%. However, another ensemble method, CatBoost Regression, did not show significantly better results than Support Vector Regression.

Moreover, a study by Shanti et al. [27] utilized several machine learning algorithms, including Support Vector Regression, K-Nearest Neighbors Regression, Random Forest Regression, Gradient Boosting Decision Trees, and Artificial Neural Networks, to predict used car prices based on their features. All these algorithms are accessible through the Python Scikit-Learn library, and the study mentions the use of a specific function from this package to fine-tune the parameters. Upon evaluation, Random Forest Regression was determined to have the best performance in terms of Mean Absolute Error (MAE), Mean Square Error (MSE), Explained Variance, and R-squared. However, Gradient Boosting Decision Trees and Artificial Neural Networks also showed competitive results, closely following the top performer.

2 Methodology

Python's Scikit-learn was chosen as the go-to statistical methods library for this study due to its wide range of both supervised and unsupervised algorithms, strong performance, and popularity among data science professionals. Its well-written documentation, vast array of related resources, and large community make it easy for the results of the study to be replicated and applied to real-world business problems. The Scikit-learn methods utilized in this study are further described in this section [23].

2.1 Ordinary Least Squares Linear Regression

Scikit-learn's `LinearRegression` fits a linear model with coefficients $\mathbf{w} = (w_1, \ldots, w_p)$ to minimize the residual sum of squares between the observed target values and the values predicted by the linear approximation. The optimization task is formulated as (1):

$$\min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2, \tag{1}$$

where:

- $\mathbf{X}$ is the matrix of input features.
- $\mathbf{w}$ is the vector of coefficients.
- $\mathbf{y}$ is the vector of observed target values.
- $\min_{\mathbf{w}}$ denotes the minimization over coefficients $\mathbf{w}$.
- $||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2$ represents the the L2 norm or the sum of the squared differences between predicted values and actual values.

The linear model prediction $\hat{y}$ for a given observation $\mathbf{x}$ is formulated as (2):

$$\hat{y}(\mathbf{w}, \mathbf{x}) = w_0 + w_1 x_1 + \ldots + w_p x_p. \tag{2}$$

To find the coefficients $\mathbf{w}$, we minimize the residual sum of squares (3):

$$\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2. \tag{3}$$

This minimization problem is equivalent to solving the following equation (4):

$$(\mathbf{X}^T\mathbf{X})\mathbf{w} = \mathbf{X}^T\mathbf{y}, \tag{4}$$

which yields the ordinary least squares solution (5):

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{5}$$

The coefficient estimates for Ordinary Least Squares rely on the presumption that the features are not correlated. For instance, When multicollinearity exists, the matrix of predictors $\mathbf{X}$ can become

close to singular. This leads to the least squares estimates being highly sensitive to random errors in the target variable, and potentially causing a large variance [19].

2.2 Support Vector Regression

Support Vector Regression (SVR) in Scikit-learn utilizes only on a subset of training data, since the cost function ignores samples whose predicted values fall within a distance ε from the actual values [26]. For a given set of training vectors $x_i \in \mathbb{R}^p$, $i = 1, \ldots, n$, and a target vector $y \in \mathbb{R}^n$, ε-SVR solves the primal problem defined as (6):

$$\min_{w,b,\zeta,\zeta^*} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}(\zeta_i + \zeta_i^*) \quad \text{subject to} \quad \begin{cases} y_i - \langle w, \phi(x_i)\rangle - b \leq \varepsilon + \zeta_i, \\ \langle w, \phi(x_i)\rangle + b - y_i \leq \varepsilon + \zeta_i^*, \\ \zeta_i, \zeta_i^* \geq 0, \quad i = 1, \ldots, n, \end{cases} , \tag{6}$$

where:

- w is the vector of coefficients,
- b is the bias term,
- ϕ denotes a feature space transformation,
- ζ and ζ^* are slack variables that measure the degree of violation of the ε-insensitivity zone,
- C is the penalty parameter of the error term, controlling the trade-off between the flatness of the regression function and the amount which exceeds the ε-insensitivity zone.

The optimization problem becomes computationally easier to solve as a Lagrange dual formulation (7):

$$\min_{\alpha,\alpha^*} \frac{1}{2}\sum_{i,j=1}^{n}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)K(x_i, x_j) + \varepsilon\sum_{i=1}^{n}(\alpha_i + \alpha_i^*) - \sum_{i=1}^{n} y_i(\alpha_i - \alpha_i^*)$$

$$\text{subject to} \quad \begin{cases} \sum_{i=1}^{n}(\alpha_i - \alpha_i^*) = 0, \\ 0 \leq \alpha_i, \alpha_i^* \leq C, \quad i = 1, \ldots, n, \end{cases} \tag{7}$$

where:

- $K(x_i, x_j) = \phi(x_i)^T\phi(x_j)$ denotes the kernel function,
- α, α^* are Lagrange multipliers.

The prediction for a new data point x is computed as (8):

$$\hat{y}(x) = \sum_{i \in SV}(\alpha_i - \alpha_i^*)K(x_i, x) + b, \tag{8}$$

where SV represents the set of support vectors, which are the data points that lie outside the ε-insensitivity zone [26].

2.3 K-Neighbors Regression

K-Neighbors Regression is a type of instance-based learning that is non-generalizing and can be used with continuous variables, since labels assigned to the query point are calculated based on the mean of k nearest neighbor labels [20].

The to find the k nearest neighbor most commonly used metric is the Euclidean distance, calculated as follows (9):

$$d(x, x') = \sqrt{\sum_{i=1}^{n}(x_i - x_i')^2}, \tag{9}$$

where x and x' are two points in the feature space, and n is the number of features. Other options for distance metrics include Manhattan and Minkowski distances.

The prediction $\hat{y}$ for a new data point x using K-Neighbors Regression is given by (10):

$$\hat{y}(x) = \frac{1}{k}\sum_{i=1}^{k} y_i, \tag{10}$$

where:

- k is the number of nearest neighbors considered for making the prediction.
- y_i are the labels of the k nearest neighbors to the query point x.

Scikit-learn offers two variations of neighbors regressors:

1. The `KNeighborsRegressor`, which bases the learning on the k nearest neighbors of each query point.
2. The `RadiusNeighborsRegressor`, which bases the learning on all neighbors within a fixed radius r of the query point.

The weights can be uniformly assigned, which is the default setting in Scikit-learn, or they can be assigned based on the distance from the query point. The latter assigns weights inversely proportional to the distance from the query point [20].

2.4 Decision Trees Regression

Decision Tree Regression in Scikit-learn implements an optimized version of the Classification and Regression Trees (CART) algorithm. This algorithm constructs decision trees for regression tasks, creating a model that predicts the value of a target variable by learning simple decision rules from the data features.

Given a training set X of size n with responses Y, a regression tree is constructed by recursively partitioning the data space and fitting a simple model (constant) within each partition. The partitioning is done in a manner that minimizes a certain criterion, typically the Mean Squared Error (MSE).

Given a set of training vectors $x_i \in \mathbb{R}^n$, $i = 1, \ldots, l$ and a label vector $y \in \mathbb{R}^l$, a decision tree recursively partitions the feature space to group samples with similar target values.

At node m, represented by Q_m with n_m samples, for each candidate split $\theta = (j, t_m)$ consisting of a feature j and threshold t_m, the data is partitioned into $Q_m^{left}(\theta)$ (11) and $Q_m^{right}(\theta)$ (12) subsets:

$$Q_m^{left}(\theta) = \{(x, y) | x_j \leq t_m\} \text{ and} \tag{11}$$

$$Q_m^{right}(\theta) = Q_m \setminus Q_m^{left}(\theta). \tag{12}$$

The impurity of node m is calculated using a loss function $H()$, and the quality of the split is measured by the function $G(Q_m, \theta)$ (13):

$$G(Q_m, \theta) = \frac{n_m^{left}}{n_m} H(Q_m^{left}(\theta)) + \frac{n_m^{right}}{n_m} H(Q_m^{right}(\theta)). \tag{13}$$

The parameters which minimize the impurity are selected (14):

$$\theta^* = \text{argmin}_\theta \, G(Q_m, \theta). \tag{14}$$

The described steps are performed recursively for subsets $Q_m^{left}(\theta^*)$ and $Q_m^{right}(\theta^*)$ until maximum allowable depth is reached.

If the target is continuous, the Mean Squared Error (MSE) criterion is often used. Alternatively, the Half Poisson deviance may also be used.

In addition, pruning, an algorithm used to avoid over-fitting, is performed using cost complexity, which is defined as (15):

$$R_\alpha(T) = R(T) + \alpha |\widetilde{T}|, \tag{15}$$

where $|\widetilde{T}|$ is the number of terminal nodes in T, $R(T)$ is the total impurity of T, and α is the complexity parameter.

The cost complexity of the entire tree T is the sum of the cost complexities of its terminal nodes (16):

$$R_\alpha(T) = \sum_{t \in \tilde{T}} R_\alpha(t). \tag{16}$$

The optimal value of α is determined through cross-validation, typically by finding the α that minimizes the cross-validated sum of errors.

The sub-tree that minimizes $R_\alpha(T)$ is chosen (17):

$$\text{Prune}(T, \alpha) = \text{argmin}_{T' \subseteq T} R_\alpha(T'), \tag{17}$$

where T' is a sub-tree of T and $R_\alpha(T')$ is the cost complexity of T' [16].

2.5 Gradient Boosting Regression

Gradient Boosting for Regression Trees (GBRT) in Scikit-learn is an additive model that is built in a stage-wise fashion. For a set of training vectors $x_i \in \mathbb{R}^n$, $i = 1, \ldots, n$, and a corresponding set of target values $y_i \in \mathbb{R}$, the prediction model $\hat{y}_i$ for input x_i is constructed as follows (18):

$$\hat{y}_i = F_M(x_i) = \sum_{m=1}^{M} h_m(x_i), \tag{18}$$

where h_m denotes the weak learners, which are decision tree regressors of fixed size, and M is the number of boosting stages, corresponding to the number of estimators parameter.

GBRT in greedy fashion adds trees to the model by fitting a new tree h_m to minimize the loss function L_m, given the current model F_{m-1} (19):

$$h_m = \arg\min_h L_m = \arg\min_h \sum_{i=1}^{n} l(y_i, F_{m-1}(x_i) + h(x_i)), \tag{19}$$

where $l(y_i, F(x_i))$ is the loss function, specified by the user through the loss parameter.

The initial model F_0 is typically the constant that minimizes the loss function; for least-squares loss, this is the mean of the target values. A first-order Taylor approximation is used to simplify the optimization of l (20):

$$l(y_i, F_{m-1}(x_i) + h_m(x_i)) \approx l(y_i, F_{m-1}(x_i)) + h_m(x_i) \left[\frac{\partial l(y_i, F(x_i))}{\partial F(x_i)} \right]_{F=F_{m-1}}. \tag{20}$$

$\left[\frac{\partial l(y_i, F(x_i))}{\partial F(x_i)} \right]_{F=F_{m-1}}$ - the derivative of the loss function with respect to its second parameter, evaluated at $F_{m-1}(x)$, is later denoted by g_i and represents the gradient of the loss function at iteration m.

After removing constant terms (21):

$$h_m \approx \arg\min_h \sum_{i=1}^{n} h(x_i) g_i. \tag{21}$$

Consequently, at each stage, the estimator h_m is fit to predict the negative gradients of the samples.

The output of the fitted tree is then adjusted to minimize the loss function L_m, with the specific update being dependent on the chosen loss function [17].

2.6 Multi-layer Perceptron Regression

The Multi-layer Perceptron (MLP) in Scikit-learn is a supervised learning algorithm which can learn a non-linear function approximation by training on a data set.

An MLP consists of at least three layers of nodes: an input layer, a hidden layer, and an output layer. Except for the input nodes, each node is a neuron that uses a non-linear activation function. The MLP utilizes a supervised learning technique called backpropagation for training the network.

Given a set of features $X = \{x_1, x_2, \ldots, x_n\}$ and a target y, the network formulates the function approximation as follows (22):

$$f(x) = W_2 g(W_1^T x + b_1) + b_2, \tag{22}$$

where:

- W_1 and W_2 are the weight matrices for the input layer to hidden layer and hidden layer to output layer, respectively.
- b_1 and b_2 are the bias vectors for the hidden layer and output layer, respectively.
- $g(\cdot)$ is the activation function applied element-wise, and is often chosen as the hyperbolic tangent function or the logistic sigmoid function (23).

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{23}$$

For regression problems, the output activation function is the identity function (i.e., no transformation is applied), and the MLP uses the Mean Squared Error (MSE) as its loss function (24):

$$Loss(\hat{y}, y, W) = \frac{1}{2n} \sum_{i=1}^{n} ||\hat{y}_i - y_i||_2^2 + \frac{\alpha}{2n} ||W||_2^2, \tag{24}$$

where:

- $\hat{y}_i$ is the predicted value for the i-th instance.
- y_i is the true value for the i-th instance.
- α is the regularization term that penalizes large weights, thereby helping to avoid over-fitting.
- $||W||_2^2$ represents the L2 norm of the weight matrix W, which is the sum of the squared weights.

The weights of the network are initialized randomly and are updated iteratively during training. After the loss is computed, a backward pass propagates weight values from outward layer to previous layers. Updated weight values are meant to decrease the loss.

While using gradient descent, the update to the weights W is calculated by subtracting the gradient of the loss function, scaled by the learning rate ϵ, from the current weights. This update is expressed as (25):

$$W^{i+1} = W^i - \epsilon \nabla_W \text{Loss}^i. \tag{25}$$

Here, i indicates the current iteration, and ϵ is the learning rate.

The gradient descent algorithm terminates when it either exceeds a preset number of iterations or the reduction in loss becomes less than a small, specified value.

Scikit-learn's MLP can be trained with different solvers for weight optimization such as Stochastic Gradient Descent (SGD), Adam, or L-BFGS [25].

2.7 Grid Search Cross-Validation

Grid Search Cross-Validation is a hyperparameter optimization technique that systematically searches through a specified parameter space for a given model [18].

The process involves defining a grid of hyperparameters and performing exhaustive search to find the optimal combination that yields the best model performance as measured by a specified evaluation metric. The search is guided by k-fold cross-validation to ensure that the model's performance is robust across different subsets of the data.

During the process of k-fold cross-validation the data set is partitioned into k sets of roughly equal size. All the samples except the first one is used to fit the model. The held-out sample is used for testing the model by predicting the values of the first set and estimating performance measures. Afterwards, first subset is returned to the training set, with the second subset being used for validation, and so on [10].

For each combination of parameters, the model is trained and evaluated using cross-validation. The performance of each parameter set is averaged over the different folds of the cross-validation. The combination that results in the highest average performance is chosen as the best hyperparameters for the model.

2.8 Min Max Scaler

The `MinMaxScaler` provided by Scikit-learn's preprocessing module scales each feature by translating it to a given range, often between zero and one [21]. The transformation is given by the following formulas (26), (27):

$$X_{\text{std}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}; \tag{26}$$

$$X_{\text{scaled}} = X_{\text{std}} \times (\max - \min) + \min. \tag{27}$$

where:

- X is the original feature value.
- $X_{\min}$ and $X_{\max}$ are the minimum and maximum values of the feature across the training set, respectively.
- X_{std} is the standard score of the feature.
- X_{scaled} is the feature value scaled to the specified range.
- min and max are the desired range of the transformed data.

2.9 Bootstrapping

Bootstrapping is a resampling technique, which takes a random sample of a data with replacement, meaning that each datapoint can be chosen again even after it has been selected for the subset. The bootstrap sample size is the same size as the original data set [10].

The bootstrapping framework employed in this study utilizes the `resample` method from Scikit-Learn. This method allows for random sampling with replacement from the dataset, creating multiple subsets for model training [24]. The approach involves the following steps:

1. **Resampling**: `resample` method is used to generate bootstrap samples of the dataset.
2. **Model Training**: A model is fitted to each of the bootstrapped samples.
3. **Prediction and Aggregation**: Predictions are made using each of the models, then their average is taken.
4. **Performance Evaluation**: Evaluation metrics are calculated based on the averaged predictions against the test set.

2.10 Evaluation Metrics

This study employs several model evaluation metrics, as detailed in the Scikit-Learn documentation [22]:

- **Mean Squared Error (MSE)**: Represents the expected value of the squares of the errors or deviations. It is calculated as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{28}$$

  where y_i is the observed value and $\hat{y}_i$ is the predicted value, while n represents the number of observations in the dataset.

- **Mean Absolute Error (MAE)**: Measures the expected value of absolute error loss. It is given by:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|, \tag{29}$$

  where y_i is the observed value and $\hat{y}_i$ is the predicted value, while n represents the number of observations in the dataset.

- **Coefficient of Determination (R-squared)**: Indicates the proportion of the variance in the dependent variable that is explained by the independent variables. Defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \tag{30}$$

  where $\bar{y}$ is the mean of observed values, y_i is the observed value and $\hat{y}_i$ is the predicted value, while n represents the number of observations in the dataset.

- **Mean Absolute Percentage Error (MAPE)**: Expresses accuracy as a percentage. Calculated by:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{\max(\epsilon, |y_i|)}, \tag{31}$$

where y_i is the observed value and $\hat{y}_i$ is the predicted value, while n represents the number of observations in the dataset. ϵ is a small constant used to avoid division by zero when the actual value y_i is zero or very close to zero, ensuring numerical stability.

3 Data

The CIS automotive data, a sample data set of CIS Automotive API, which can be found on Kaggle, was chosen as the data set for this study. It initially contained 2.4 million daily data points from approximately 1,200 dealers in Illinois, USA, covering the period from June 2018 to June 2020. Each data point includes car features, the date a car was last seen at the dealership and its asking price at that time. An assumption can be made that asking price, at the time the car was last seen, was the actual price at which the car was sold to a customer. The steps taken to prepare the data set for model training and the exploratory analysis of the data are further described in this section.

3.1 Data Preparation

In the initial stages of data preparation, each categorical variable's distinct values were thoroughly checked and adjusted if needed. One example of such adjustments could be records with a car body class of `Sedan/Saloon`. Such cases were reclassified and assigned together with a another body class - `Sedan`. Another example involves typographical errors in model names. Fixing such mistakes was crucial in order to preserve as many data points as possible and to simplify the encoding of categorical variables.

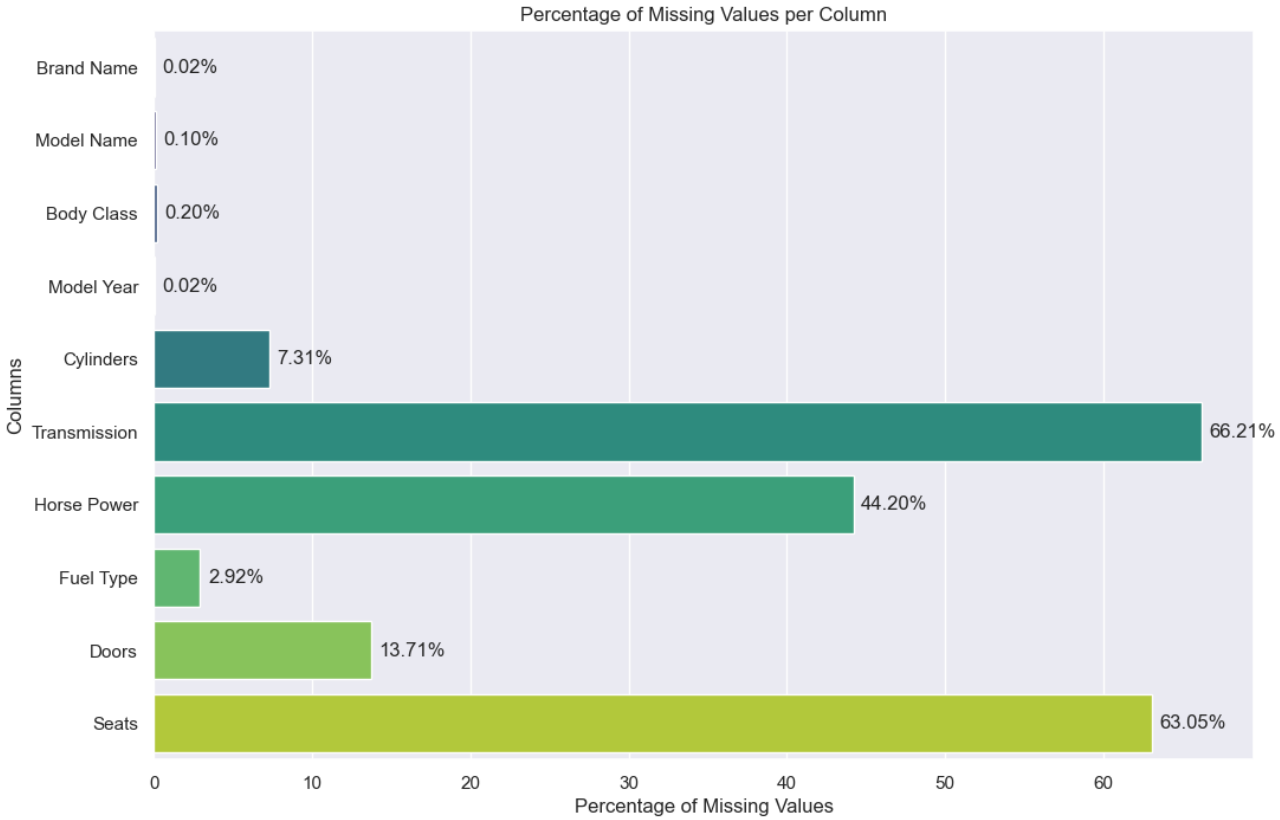

Figure 1: Percentage of Missing Values per Column

As it is observed in Figure 1, some columns were missing more values compared to others. Specifically, `Transmission` column was excluded because of significant number of missing entries. In addition,

the existing values lacked variability - majority of existing values were `Automatic` - which further justified its removal from further analysis. However, some of the missing values were easier to infer - for example, it is reasonable to assume that a sedan has 5 seats. After cleaning the data which was possible to save, the rest of the rows with at least one missing value was removed.

Furthermore, observations where price values were 1.5 times the interquartile range above the third quartile or below the first quartile, calculated for groups based on car brand, model and age at sale, were also removed. Finally, the cleaned data set consisted of 561,786 data points.

In addition, a new variable `Age at Sale` was constructed by subtracting a year of the model from the date a car was last seen at the dealership. Summary statistics of the initial set of categorical and numeric variables are shown in Table 1 and Table 2.

	Date	**Brand Name**	**Model Name**	**Body Class**	**Fuel Type**
count	561,786	561,786	561,786	561,786	561,786
unique	731	35	289	8	4
top	2018-08-06	FORD	RAV4	SUV	Gasoline
freq	3,969	90,322	24,696	318,960	551,020

Table 1: Summary Statistics of the Categorical Variables

	Price	**Mileage**	**Age at Sale**	**Cylinders**	**Horse Power**	**Doors**	**Seats**
count	561,786	561,786	561,786	561,786	561,786	561,786	561,786
mean	31,117.95	5,401.47	0.50	4.80	229.78	4.15	5.35
std	13,562.52	13,581.50	0.92	1.23	76.67	0.52	0.92
min	999.00	0.00	0.00	0.00	72.00	2.00	2.00
25%	21,958.00	0.00	0.00	4.00	175.00	4.00	5.00
50%	28,249.00	0.00	0.00	4.00	200.00	4.00	5.00
75%	37,147.00	5.00	1.00	6.00	287.00	4.00	5.00
max	469,990.00	413,305.00	29.00	12.00	815.34	5.00	8.00

Table 2: Descriptive Statistics of Numeric Features

Moreover, it was necessary to consider collinearity among independent variables. As it is visible in the correlation heat map in Figure 2, the variables `Cylinders` and `Horse Power` showed a correlation of 0.87. Therefore, `Cylinders` was not included in further analysis, since `Horse Power` provided greater variability.

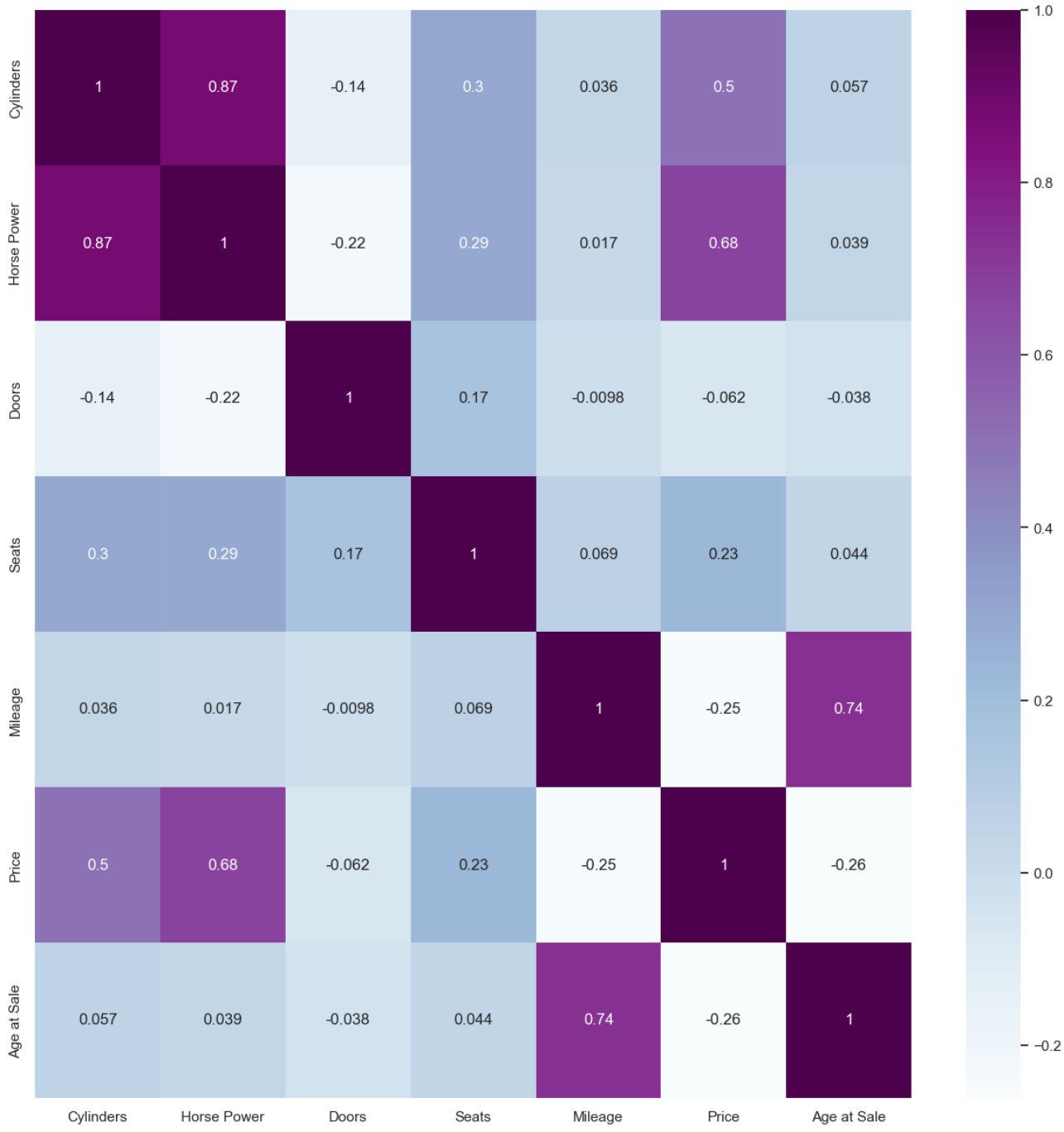

Figure 2: Correlation Heat Map of Numeric Variables

3.2 Data Preprocessing

Before training the models, the data had to be properly processed. All categorical variables were encoded using Scikit-learn's `OneHotEncoder` class, which creates a binary dummy variable for each unique category within a categorical variable. Even though this method increase data dimensionality, in our case, the number of variables did not overly increase the training time of the models.

In addition, numeric variables were processed using Scikit-learn's `MinMaxScaler` class. Numeric variables were scaled in the range of 0 and 1, to minimize the the differences in units and reduce the bias caused by varying scales of the variables. Another reason why feature scaling was implemented is because some algorithms, for example those that use gradient descent as an optimization technique, require feature scaling to ensure faster convergence.

Finally, to account for potential variations in car prices over time, a monthly dummy variable was included in the final set of the explanatory variables. `Month` variable was created using the date the car was last seen at the dealership. The data used in this study does not qualify as time series data because each unique vehicle, identified by its VIN code, appears on average in only 1.96 observations. Consequently, it is not feasible to construct time series data where a collection of observations of the same vehicle are tracked over multiple periods of time. However, even though the data set used in this study more closely resembles pooled cross-sectional data, there is a possibility that time-related or seasonal trends could influence car prices.

4 Modeling

A modeling framework was introduced to further reduce the computing time of modeling and to decrease the dimensionality of the data. In the beginning stage of the framework, data set is split into subsets according to car brands. All the steps of the framework are performed separately for each car brand. This approach helps to reduce number of data points used to train the models. Also, binary dummy variables created using `Car Model` are specific to each car brand, which further reduces dimentionality of the data set. Modeling framework consists of:

- **Data Splitting:** The data points are randomly assigned to an 80/20 train-test split.
- **Categorical Variable Encoding:** Categorical variables are encoded into binary dummy variables using `OneHotEncoder`.
- **Numeric Variable Scaling:** Numeric variables are scaled using `MinMaxScaler`.
- **Hyperparameter Optimization:** Hyperparameters are optimized using grid search cross-validation, where applicable, for each model.
- **Model Training:** The model is trained on the training dataset using the optimized hyperparameters.
- **Model Evaluation:** The model is evaluated using the testing data set.

The top 3 car brands, selected based on the number of data points available in our data set, were used to train and test a set of modeling methods. All of the models were evaluated before and after the parameter tuning. This section includes the description of the steps taken to optimize the performance of each of the modeling techniques.

4.1 Ordinary Least Squares Regression

Implementing ordinary least squares regression serves as a great starting point of the experiment aimed at finding the best method for predicting a car price. OLS is unique among the methods tested in this study as it does not require any parameter tuning. The linear regression model was trained on a training set of explanatory variables with corresponding values of the response variable `Price`.

At first the method was tested with default numeric variables and encoded categorical variables. For instance, when training the OLS model on Ford training data set, Mean Absolute Percentage Error (MAPE) of 0.13 was achieved for both train and out-of-sample test data sets. Figure 3 shows the OLS model predicted values plotted against the actual values of the response variable from the test set of Ford.

In addition, because of differences in scales of numeric independent variables - for example, `Mileage` is in hundreds of thousands while `Horse Power` is in hundreds - the OLS model was tested with scaled numeric independent variable. However, this change did not influence the model performance. As it is visible in Figure 4, MAPE for both train, and out-of-sample test data sets remained the same.

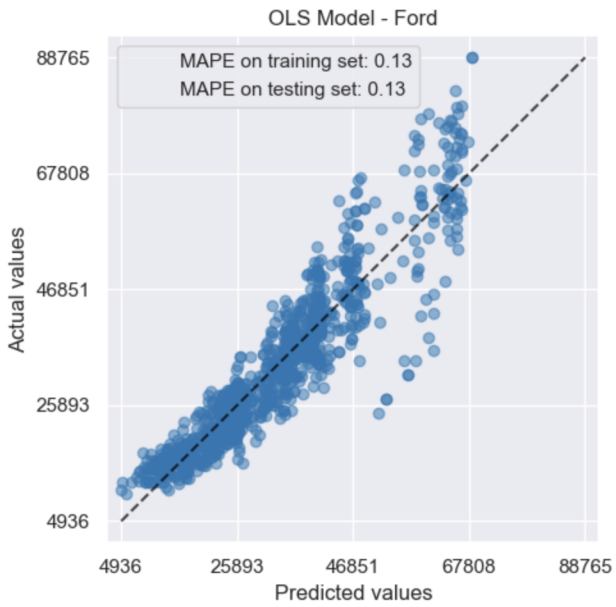

Figure 3: OLS Regression - Actual vs. Predicted

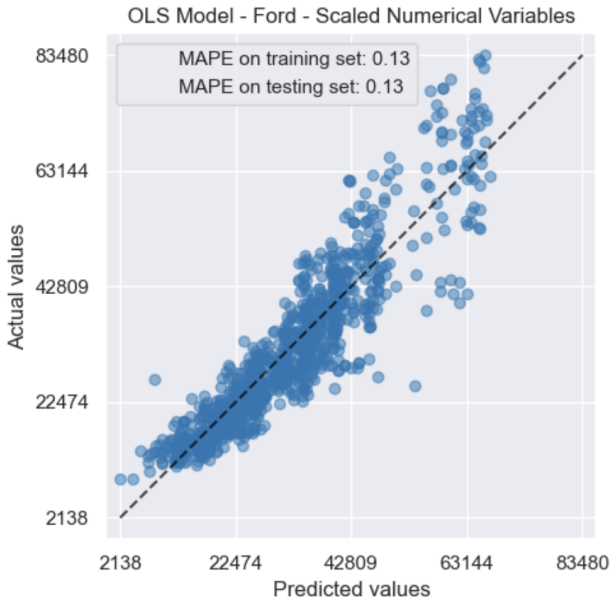

Figure 4: OLS Regression with Scaled Numeric Independent Variables - Actual vs. Predicted

4.2 Support Vector Regression

The second candidate of the experiment was Support Vector Regression. The model was trained on a test data set with scaled numeric independent variables. A linear kernel was chosen first, due to its faster performance, especially with the large size of our dataset. However, as shown in Figure 5, after being trained on the Ford training dataset with default parameters, it performed worse than the OLS regression, achieving only an out-of-sample MAPE of 0.18. Support Vector Regression with a Gaussian kernel and default parameters did even worse, with an out-of-sample MAPE of 0.31 (Figure 6). Notably, training the SVR with a linear kernel on the Ford dataset took 0.06 seconds, while with a Gaussian kernel, it took 224.2 seconds. The performance of the SVR with a Gaussian kernel might have posed a real issue when tuning hyperparameters using Grid Search Cross-Validation, as it is an exhaustive search requiring multiple iterations of model training. Therefore, only the linear kernel was used further on.

Epsilon, a parameter specifying the epsilon-tube within which the training loss function does not penalize errors, and the regularization parameter C, were chosen as the parameters to be tuned using Grid Search Cross-Validation. The parameter grid was constructed from epsilon values ranging from 0 to 1 and C values from 0.1 to 1000. An epsilon of 1 and a C of 100 were chosen as the optimal parameter values for the Ford dataset, based on the lowest MAPE value. However, the overall result did not prove to be any better than the one produced by Ordinary Least Squares Regression (Figure 7) [26].

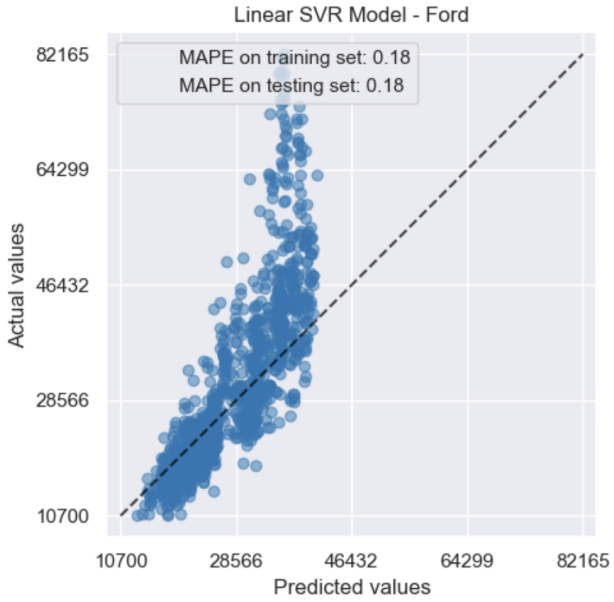

Figure 5: Linear SVR - Actual vs. Predicted

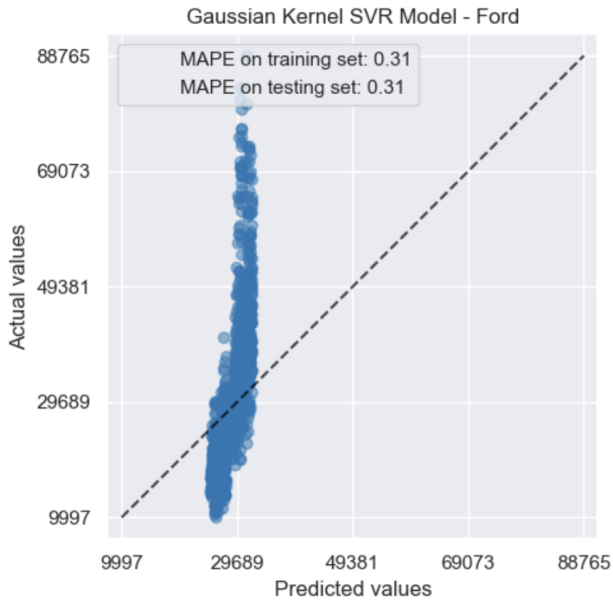

Figure 6: Gaussian Kernel SVR - Actual vs. Predicted

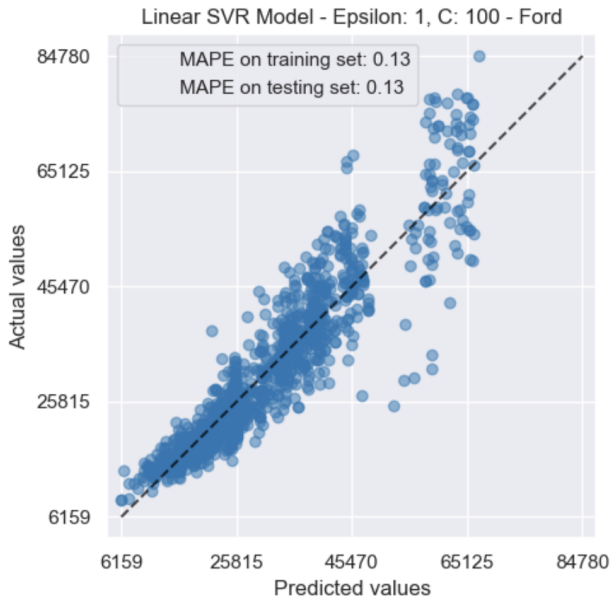

Figure 7: Linear SVR Tuned - Actual vs. Predicted

4.3 K-Neighbors Regression

The K-Neighbors Regression model was trained on a test dataset with scaled numeric independent variables. After training on the Ford training dataset using default parameters, the model already showed a better result than the Ordinary Least Squares regression, achieving an out-of-sample MAPE of 0.1, as shown in Figure 8.

Parameters 'n neighbors', 'weights', and 'metric' were selected for tuning using Grid Search Cross-Validation. The parameter 'n neighbors' defines the number of nearest neighbors to consider, where

a too small number might lead to over-fitting, and a too large number could prevent the model from generalizing effectively. Furthermore, 'weights' determines the weights given to the contributions of the neighbors. 'Uniform' weighs all neighbors equally, while 'distance' gives greater weights to closer neighbors, potentially enhancing the model's performance on unevenly distributed data. Lastly, 'metric' specifies the distance metric used for finding nearest neighbors. An 'n neighbors' of 15, distance-based 'weights', and the 'Manhattan' distance metric were identified as the optimal parameter values for the Ford dataset, based on the lowest MAPE value. This further improved the K-Neighbors regression out-of-sample MAPE to 0.09, as depicted in Figure 9 [20].

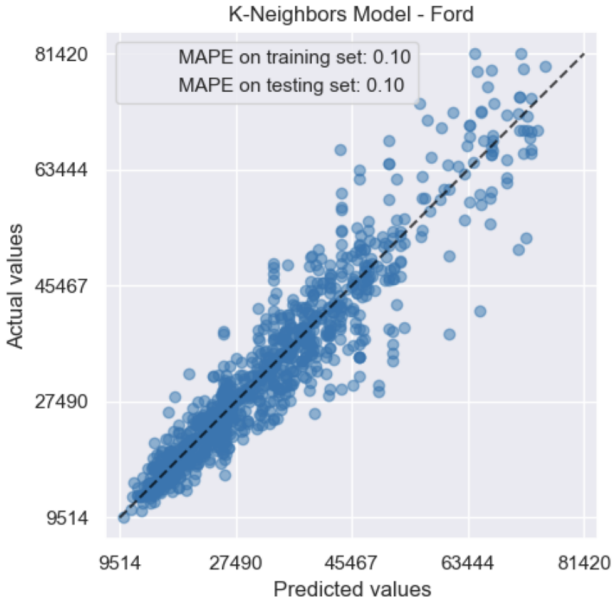

Figure 8: K-Neighbors Regression - Actual vs. Predicted

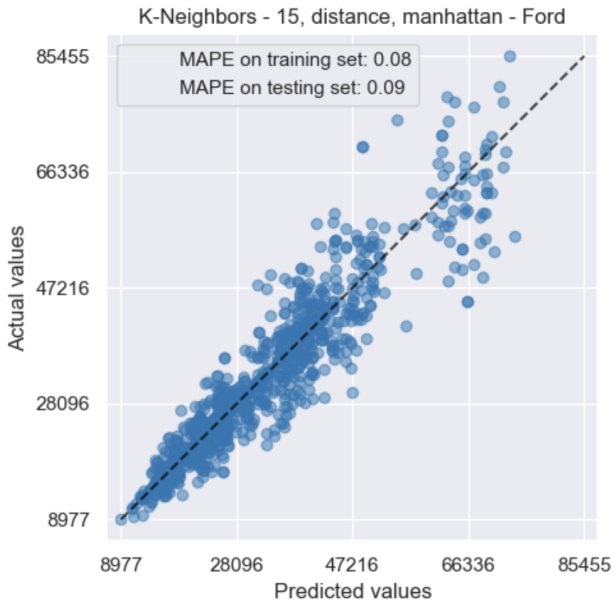

Figure 9: K-Neighbors Regression Tuned - Actual vs. Predicted

4.4 Decision Trees Regression

Consistent with findings in the related literature [27] [28], Decision Tree Regression emerged as the leading methodology after being trained on the Ford dataset with scaled numeric independent variables. Without hyperparameter optimization, this model demonstrated high performance compared to other methods, achieving an out-of-sample MAPE of 0.09, as illustrated in Figure 10.

Parameters 'max depth', 'min samples split' and 'min samples leaf' were chosen for further tuning using Grid Search Cross-Validation. 'Max depth' defines maximum depth of the tree, with deeper tree being able to model more complex patters. However, a deep tree could also lead to over-fitting. Minimum number of samples required to split an internal node and minimum number of samples required to be at a leaf node both control for over-fitting preventing the tree from getting too granular. However, in our case, parameter tuning did not yield significant improvement in model performance, since the default parameter values turned out to also be the optimal ones. As depicted in Figure 11, the out-of-sample MAPE remained unchanged after parameter tuning when tested with the Ford test dataset. The optimal maximum depth of the tree was found to be 'None', meaning that nodes are

expanded until all leaves are pure or until all leaves contain less than the defined minimum number of samples required to split an internal node, without capping maximum tree depth to a predefined number. Moreover, the optimal value of 'min samples split' was found to be 2, which is equal to the default value of this parameter. Finally, the optimum minimum number of samples required to be at a leaf node was found to be 1, which was also equal to the default value of the parameter [16].

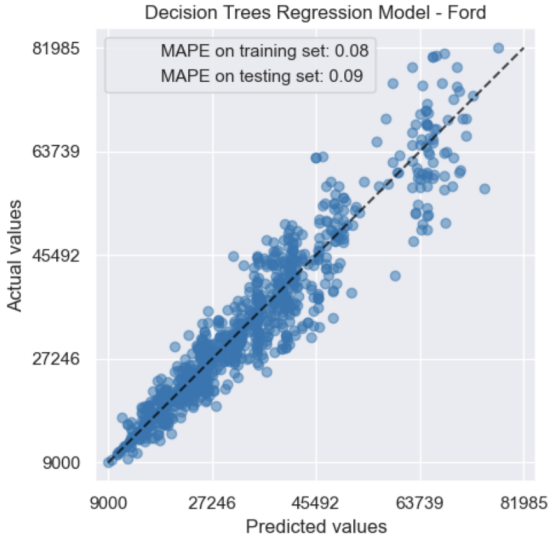

Figure 10: Decision Trees Regression - Actual vs. Predicted

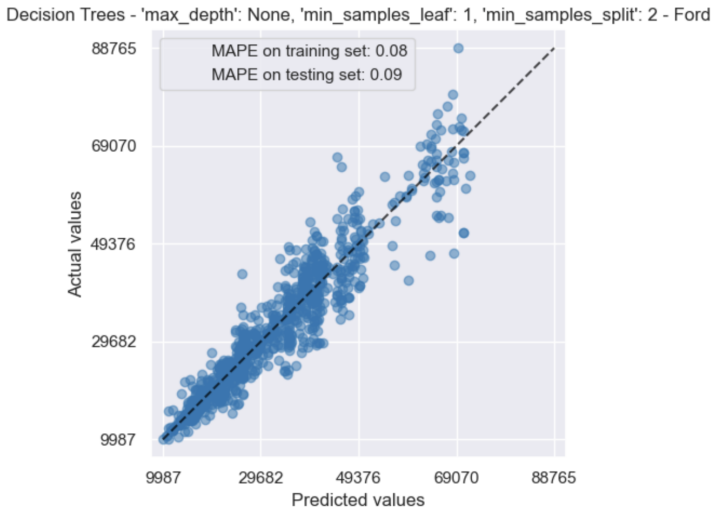

Figure 11: Decision Trees Regression Tuned - Actual vs. Predicted

4.5 Gradient Boosting Regression

Another method, which was deemed to be promising by the related literature [11], was Gradient Boosting Regression. A member of the ensemble methods family, trained on the Ford dataset with scaled numeric independent variables, with default parameters achieved an out-of-sample MAPE of 0.11, as shown in Figure 12.

For further tuning via Grid Search Cross-Validation, the parameters selected were 'n estimators', 'learning rate', and 'max depth'. 'Max depth' defines maximum depth of each tree. The logic is similar as for the Decision Trees Regression, where deeper trees are able to model more complex patters, although with increasing risk of over-fitting. Another parameter 'learning rate' affects the contribution of each tree to the final model. Lastly, 'n estimators' determines how many sequential trees are going to be modeled. After conducting 5-fold Grid Search Cross-Validation with the Ford dataset, the model with parameters optimized for the lowest MAPE score achieved an out-of-sample MAPE of 0.09, matching the performance of both the non-tuned and tuned Decision Tree Regression, as well as the tuned K-Neighbors Regression, as depicted in Figure 13. The optimal learning rate was found to be 0.1, which is also a default value of this parameter. The optimal maximum depth of each tree was found to be 10, which was an increase from a default value of 3. It could mean that our data requires a model which captures more complex relationships by allowing more splits, also capturing more information about the data. Lastly, the optimal 'n estimators' value was found to be 300, which is 3 times more than the default value of 100. A higher number of boosting stages can lead to better

learning from the data, up to a certain threshold beyond which the improvement may plateau [17].

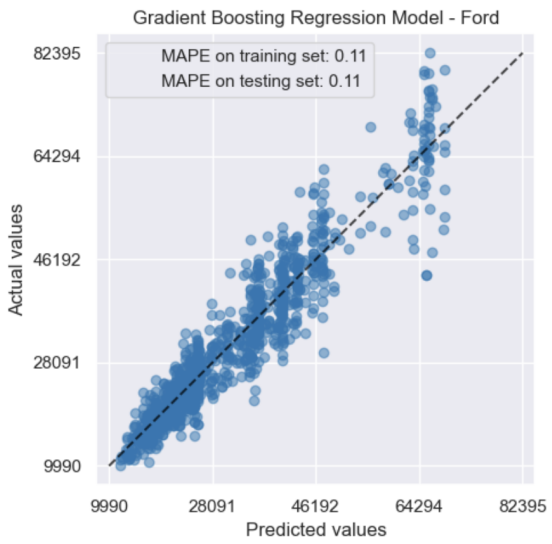

Figure 12: Gradient Boosting Regression - Actual vs. Predicted

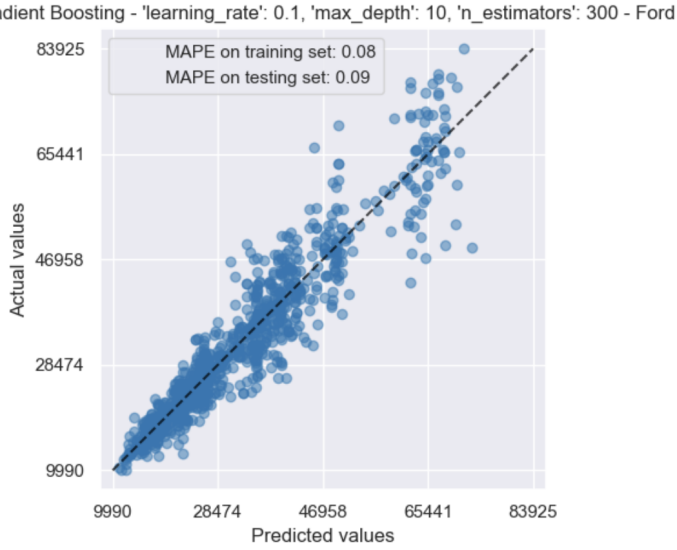

Figure 13: Gradient Boosting Regression Tuned - Actual vs. Predicted

4.6 Multi-layer Perceptron Regression

Lastly, to represent supervised neural network models, Multi-layer Perceptron (MLP) Regression was also included in this study. When trained on the Ford dataset with scaled numeric independent variables, with default parameters MLP Regression achieved an out-of-sample MAPE of 0.13, as shown in Figure 14. Throughout the course of the study, stochastic gradient-based optimizer 'adam' was kept as a preferred solver because of its strong performance with relatively large datasets [25].

Multi-layer Perceptron Regression emerged as one of the slowest methods evaluated in this study. For comparison, training MLP Regression with default parameters on Ford dataset took 93.89 seconds, while Decision Trees Regression was able to finish the same task in 0.16 seconds. In addition, MLP Regression involves multiple hyperparameters that need to be tuned to achieve the optimal results. Given that Grid Search Cross-Validation demands training and testing the model for each combination of values within the hyperparameter grid, this task can be both time-consuming and computationally intensive.

During the training phase of Grid Search Cross-Validation, the optimization of the algorithm was encountering convergence issues. To address this, the initial learning rate was increased to 0.1 to ensure more substantial step sizes for efficient convergence. Additionally, the maximum number of iterations was doubled from 200 to 400. 'Early stopping' along with a 'validation fraction' of 0.1 were also implemented to enhance the algorithm's speed. Early stopping allows for the termination of training when there is no improvement in the validation score, while the 'validation fraction' specifies the portion of the training dataset reserved for validation purposes. Furthermore, tuning focused on combinations of 'hidden layer sizes' and 'alpha'. Different 'hidden layer sizes' configurations can capture varying levels of data complexity, whereas 'alpha', the L2 regularization term, aids in preventing over-fitting by penalizing large weights. After executing 5-fold Grid Search Cross-Validation on the Ford dataset, the

model with parameters optimized for the lowest MAPE score attained an out-of-sample MAPE of 0.11, as shown in Figure 15. The optimal hidden layer size was found to be (80, 80), meaning that network architecture was changed from the default value (100,), having one hidden layer with 100 neurons, to two hidden layers with each of them having 80 neurons. In addition, after parameter tuning, 'alpha' was increased from a default value of 0.0001 to 0.2. The additional hidden layer helped the model to capture more complex patterns, while the increased 'alpha' value contributed to mitigating the risk of over-fitting [25].

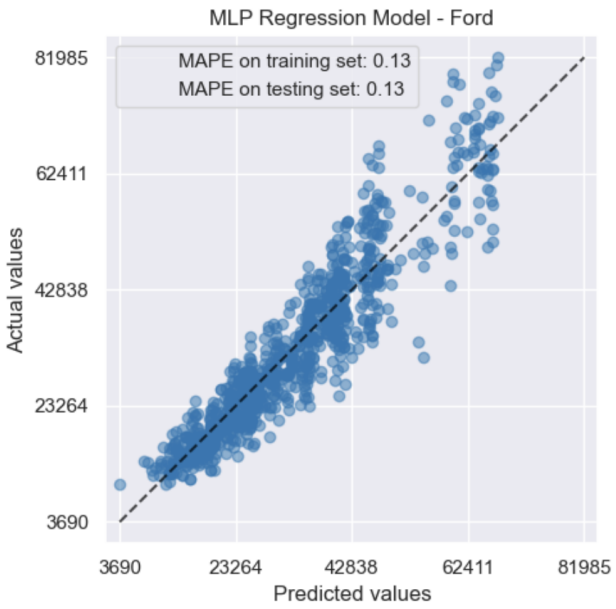

Figure 14: MLP Regression - Actual vs. Predicted

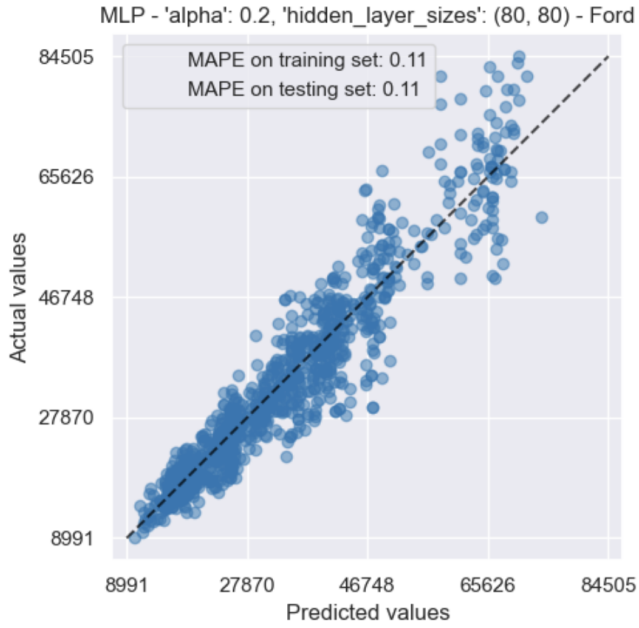

Figure 15: MLP Regression Tuned - Actual vs. Predicted

5 Results

Datasets of top 3 car brands, selected based on the number of data points available in our data set, were used to train and test the set of modeling methods. Models were evaluated based on Mean Squared Error (MSE), Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE), calculated using test set of actual dependent variable `Price` values and values predicted using the estimated model coefficients and corresponding actual independent variable values from the test dataset. In addition, model training was timed to compare whether an increase in complexity and time consumption of models would lead to corresponding improvements in results.

The results for models with default parameters are shown in Table 3. Decision Trees Regression showed the best results across all of the car brands and evaluation metrics. In addition, it was one of the faster methods together with OLS Regression, Support Vector Regression and K-Neighbors Regression, all taking less than 1 second to train the model for each of the car brands. Gradient Boosting Regression was multiple times slower, taking from 1.83 seconds to 3.78 seconds to finish the same task. It still may not seem much, but computation time can quickly add up when tuning the parameters, especially if there are more than a couple of parameters and a researcher cannot test only a small range of parameter values. Multi-layer Perceptron Regression took the longest with training times taking from 65.07 seconds to 97.37 seconds depending on the size of the dataset it was trained on. However, these longer training times could not be justified by the results since, without hyperparameter tuning, both Gradient Boosting Regression and Multi-layer Perceptron Regression performed worse than a way faster Decision Trees Regression.

Table 4 contains the model results achieved after the hyperparameter tuning using the Grid Search Cross-Validation. Parameter tuning significantly enhanced the performance of Support Vector Regression, while other methods also demonstrated improvements, albeit less pronounced. However, Decision Trees Regression again showcased the best results across all of the car brands and evaluation metrics. Nonetheless, after parameter tuning, Gradient Boosting Regression and K-Neighbors Regression achieved the results close to the ones of Decision Trees Regression.

Brand Name	Model	MSE Test	MAE Test	MAPE Test	Time
FORD	OLS Regression	30264987.2	4004.2	0.133004	0.06 s
FORD	Support Vector Regression	107771858.6	6582.9	0.182648	0.05 s
FORD	K-Neighbors Regression	23955441.9	3378.0	0.104757	0.004 s
FORD	Decision Trees Regression	19665572.7	2952.0	0.090332	0.12 s
FORD	Gradient Boosting Regression	22819216.7	3482.8	0.111840	3.78 s
FORD	MLP Regression	29240839.2	3916.0	0.127523	97.37 s
TOYOTA	OLS Regression	15016534.3	3005.3	0.107781	0.04 s
TOYOTA	Support Vector Regression	44105432.6	4906.0	0.174703	0.02 s
TOYOTA	K-Neighbors Regression	11775108.4	2523.9	0.084217	0.003 s
TOYOTA	Decision Trees Regression	9390518.5	2165.3	0.070827	0.09 s
TOYOTA	Gradient Boosting Regression	11331456.1	2571.3	0.088143	2.95 s
TOYOTA	MLP Regression	15338652.1	3031.8	0.109121	73.73 s
JEEP	OLS Regression	20405365.8	3587.8	0.122014	0.02 s
JEEP	Support Vector Regression	38781591.1	4731.0	0.151965	0.02 s
JEEP	K-Neighbors Regression	20327010.6	3418.5	0.113840	0.003 s
JEEP	Decision Trees Regression	17151158.7	3035.7	0.100366	0.06 s
JEEP	Gradient Boosting Regression	18200236.6	3349.9	0.112633	1.83 s
JEEP	MLP Regression	20239487.5	3572.8	0.120990	65.07 s

Table 3: Models with Default Parameters Comparison Results

Brand Name	Model	MSE Test	MAE Test	MAPE Test
FORD	OLS Regression	30264987.2	4004.2	0.133004
FORD	Support Vector Regression	31200452.8	3985.9	0.127585
FORD	K-Neighbors Regression	20139511.8	2983.2	0.090863
FORD	Decision Trees Regression	19628112.0	2951.4	0.090327
FORD	Gradient Boosting Regression	19259130.4	2964.0	0.091080
FORD	MLP Regression	21925074.2	3407.4	0.109233
TOYOTA	OLS Regression	15016534.3	3005.3	0.107781
TOYOTA	Support Vector Regression	15169016.2	2982.6	0.106624
TOYOTA	K-Neighbors Regression	9808379.5	2216.1	0.072400
TOYOTA	Decision Trees Regression	9389851.4	2165.6	0.070851
TOYOTA	Gradient Boosting Regression	9171865.3	2161.0	0.070799
TOYOTA	MLP Regression	11171889.2	2552.2	0.085779
JEEP	OLS Regression	20405365.8	3587.8	0.122014
JEEP	Support Vector Regression	20801136.4	3557.2	0.118838
JEEP	K-Neighbors Regression	17502718.0	3062.2	0.101432
JEEP	Decision Trees Regression	17162533.6	3036.5	0.100424
JEEP	Gradient Boosting Regression	16779309.4	3029.4	0.100226
JEEP	MLP Regression	18323380.0	3338.1	0.111565

Table 4: Tuned Models Comparison Results

In addition, to reduce the variance in model predictions and to assess the reliability of the most effective model identified in this study, bootstrapping was employed in the training and testing of Decision Trees Regression. The re-sampling with replacement was performed a 1000 times with Ford, Toyota and Jeep datasets separately. Decision Trees Regression with optimal parameters recorded after

tuning process was fitted and response variable values were predicted for each of the bootstrapping iteration. Also, an average of each of the predicted value was calculated out of the 1000 bootstrapping samples. The averaged predicted values and test set of actual dependent variable values were used to calculate the evaluation metrics which are visible in Table 5. Even after bootstrapping, Decision Trees Regression performed on par with the results recorded in the earlier stages of the study. For example, after trained and tested on Ford dataset, Decision Trees Regression with tuned parameters achieved MAPE of 0.09 while after bootstrapping with 1000 samples, the same model achieved MAPE of 0.091.

Moreover, in Table 5 it is shown that standard deviation of the dependent variable `Price` was very similar for both train and test samples of all the datasets (Ford, Toyota and Jeep). Since R-squared statistic shows the proportion of the variance in the dependent variable that is predictable from the independent variables, we could see that some of the variance of Price was still left unexplained. As seen in Table 5, R-squared differs between car brands, with our model explaining greater portion of variance in prices of Ford and Toyota vehicles than Jeep automobiles. This could mean that there could exists independent variables, not included in the study, which affect Jeep prices more than they affect Ford or Toyota prices.

Brand Name	**MAPE Test**	**R2 Test**	**Train Price SD**	**Test Price SD**
FORD	0.091326	0.906575	14262.5	14367.2
TOYOTA	0.071827	0.876693	8738.7	8634.9
JEEP	0.101511	0.797351	9158.9	9106.4

Table 5: Bootstrapped Decision Trees Regression Results

Lastly, feature importance was calculated using the best performing model of the study - Decision Trees Regression. Feature importance is calculated as the reduction in a model's error brought by that feature. It's a measure of how much a feature decreases the prediction error, and in our case - the sum of squared errors. In other words, it quantifies the contribution of each feature to the accuracy of the model, helping to identify which features are most predictive for the target variable [16]. Tables 6, 7, and 8 display the top 10 most significant features for the Decision Trees Regression model, specific to the Ford, Toyota, and Jeep data sets, respectively. Across all three car brands, `Horse Power` emerged as the most influential feature. `Age at Sale` was another important feature, ranking as the second most important for the models trained on Toyota and Jeep data sets, and the third most important for the the model trained on Ford data set. The importance of other independent variables differed across car brands. For example, `Mileage` was more influential in predicting prices for Toyota and Jeep compared to Ford. In addition, monthly dummy variables, which were included to account for potential variations in car prices over time, did not exhibit significant influence on car prices - at least within the time period observed in this study. Only the January monthly dummy variable featured among the top 10 most important features for models trained on Ford and Jeep data sets.

Feature	Importance
Horse Power	0.682890
Fuel Type Gasoline	0.099194
Age at Sale	0.082284
Seats	0.077282
Mileage	0.028278
Body Class Sedan	0.002610
Model Name Mustang	0.002127
Doors	0.001970
Month 1	0.001821
Model Name Explorer	0.001649

Table 6: Decision Trees Regression - Feature Importance - Ford

Feature	Importance
Horse Power	0.482624
Age at Sale	0.149611
Mileage	0.102822
Model Name Corolla	0.097187
Model Name Sequoia	0.034063
Body Class SUV	0.024808
Model Name Avalon	0.021363
Doors	0.018573
Model Name LAND CRUISER	0.014402
Model Name Yaris iA	0.004442

Table 7: Decision Trees Regression - Feature Importance - Toyota

Feature	Importance
Horse Power	0.756584
Age at Sale	0.162489
Mileage	0.042354
Model Name Wrangler JK	0.005293
Model Name Renegade	0.005074
Model Name Wrangler	0.004908
Seats	0.004600
Fuel Type Gasoline	0.001946
Month 1	0.001698
Doors	0.001533

Table 8: Decision Trees Regression - Feature Importance - Jeep

6 Conclusion

In this study, the CIS automotive dataset, a subset of the CIS Automotive API with 2.4 million daily data points from around 1,200 dealers in Illinois, USA, spanning June 2018 to June 2020, was utilized to assess various regression models. The data cleaning was conducted using the R programming language, followed by pre-processing which included encoding categorical variables and scaling numerical ones. The analysis involved training and testing models on datasets for three car brands - Ford, Toyota, and Jeep. The assessed models included OLS Regression, Support Vector Regression, K-Neighbors Regression, Gradient Boosting, and MLP Regression. Moreover, parameter tuning was applied using Grid Search Cross-Validation to enhance results beyond those obtained with default parameters. The pre-processing, modeling, and tuning were executed using Python and the Scikit-learn library.

After concluding all of the previously mentioned steps, it was found that Decision Trees Regression, was both the fastest and best-performing method for predicting a car price given a set of features associated with a particular car with both default and tuned hyperparameters between all three different car brand datasets. High speed performance of Decision Trees Regression from the Scikit-learn library makes it possible to train and test multiple models on a vast amount of data in a timely manner. The comprehensively documented Scikit-learn library, alongside its extensive resources and community support, further facilitates the replication and real-world application of this study's findings. In addition, the robustness of study results, achieved with Decision Trees Regression, was tested using bootstrapping - training models on 1000 re-sampled data samples and evaluating their averaged predictions. The results achieved with bootstrapping were on par with results achieved with single train-test sample, which further enforced the robustness of Decision Trees Regression.

While `Horse Power` and `Age at Sale` emerged as the most important features when using Decision Trees Regression, monthly dummy variables, intended to reflect time-based car price fluctuations, had minimal impact. However, the results of this study could still be improved upon by including more independent variables, which could explain even more variance in price of a vehicle. Such independent variables could include trim level or number of previous accidents. Since a significant portion of data in this study was excluded due to quality issues or missing values, utilizing a high-quality in-house dataset collected by a car sales business could potentially yield even better results with the models tested in this study. Moreover, having a longer time series of data together with macroeconomic indicators, such as inflation, could provide deeper insights into the variations in car prices.

References

[1] Bartlett, J. Cars are expensive. here's why and what you can do about it.. *Consumer Reports.*, https://www.consumerreports.org/cars/buying-a-car/people-spending-more-on-new-cars-but-prices-not-necessarily-rising-a3134608893/

[2] Chandak, A., Ganorkar, P., Sharma, S., Bagmar, A. & Tiwari, S. Car price prediction using machine learning. *International Journal Of Computer Sciences And Engineering.* **7**, 444-450 (2019)

[3] CNN, B. How covid has changed the way used cars are sold | CNN business. *CNN.* (2021,3), https://edition.cnn.com/2021/03/09/success/online-used-car-shopping-feseries/index.html

[4] Cui, B., Ye, Z., Zhao, H., Renqing, Z., Meng, L. & Yang, Y. Used car price prediction based on the iterative framework of xgboost+lightgbm. *Electronics.* **11**, 2932 (2022)

[5] Desai, P. & Purohit, D. Leasing and selling: Optimal Marketing Strategies for a durable goods firm. *Management Science.* **44**, 19-34 (1998)

[6] Dunn, J. & Leibovici, F. Federal Reserve Economic Data: Your trusted data source since 1991. *FRED Blog.*, https://fredblog.stlouisfed.org/2022/10/whats-been-driving-the-rise-in-auto-prices-since-covid/

[7] Du, J., Xie, L. & Schroeder, S. practice prize paper—pin optimal distribution of auction vehicles system: Applying price forecasting, elasticity estimation, and genetic algorithms to used-vehicle distribution. *Marketing Science.* **28**, 637-644 (2009)

[8] Ellencweig, B., Espel, P., Jovanovic, S., Tschauner, B. & Quidwai, K. Data and analytics in the driver's seat of the used-car market. *McKinsey & Company.* (2023,8), https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/data-and-analytics-in-the-drivers-seat-of-the-used-car-market

[9] Kooreman, P. & Haan, M. Price anomalies in The used car market. *De Economist.* **154**, 41-62 (2006)

[10] Kuhn, M., & Johnson, K. (2013). *Applied Predictive Modeling.* Springer. Available at https://link.springer.com/book/10.1007/978-1-4614-6849-3.

[11] Lessmann, S. & Voß, S. Car resale price forecasting: The impact of regression method, private information, and heterogeneity on forecast accuracy. *International Journal Of Forecasting.* **33**, 864-877 (2017)

[12] Liu, E., Li, J., Zheng, A., Liu, H. & Jiang, T. Research on the prediction model of the used car price in view of the PSO-Gra-BP Neural Network. *Sustainability.* **14**, 8993 (2022)

[13] Longani, C., Prasad Potharaju, S. & Deore, S. Price prediction for pre-owned cars using Ensemble Machine Learning Techniques. *Recent Trends In Intensive Computing.* (2021)

[14] Purohit, D. Exploring the relationship between the markets for new and used durable goods: The case of automobiles. *Marketing Science*. **11**, 154-167 (1992)

[15] Rifkin, R. M., & Lippert, R. A. Notes on Regularized Least Squares. Technical Report MIT-CSAIL-TR-2007-025, CBCL-268, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1, 2007. Available at http://cbcl.mit.edu/publications/ps/MIT-CSAIL-TR-2007-025.pdf. Accessed 16-December-2023.

[16] Scikit-Learn Developers. Decision Trees: Regression. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/tree.htmlregression. Accessed 16-December-2023.

[17] Scikit-Learn Developers. Ensemble Methods. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/ensemble.html. Accessed 16-December-2023.

[18] Scikit-Learn Developers. Hyperparameter Tuning with GridSearchCV. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/grid$_search.html.Accessed16 - December - 2023$.

[19] Scikit-Learn Developers. Linear Models. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/linear$_model.html.Accessed16 - December - 2023$.

[20] Scikit-Learn Developers. Nearest Neighbors. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/neighbors.html. Accessed 16-December-2023.

[21] Scikit-Learn Developers. Preprocessing data: Scaling. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/preprocessing.htmlpreprocessing-scaler. Accessed 16-December-2023.

[22] Scikit-Learn Developers. Regression metrics. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/model$_evaluation.htmlregression - metrics$.

[23] Scikit-Learn Development Team. Scikit-Learn User Guide. Available at https://scikit-learn.org/stable/user$_guide.html.Accessed16 - December - 2023$.

[24] Scikit-Learn Developers. *sklearn.utils.resample*. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/generated/sklearn.utils.resample.html.

[25] Scikit-Learn Developers. Supervised Learning: Neural Networks. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/neural$_networks_supervised.html.Accessed16 - December - 2023$.

[26] Scikit-Learn Developers. Support Vector Machines. Scikit-Learn Documentation. Available at https://scikit-learn.org/stable/modules/svm.html. Accessed 16-December-2023.

[27] Shanti, N., Assi, A., Shakhshir, H. & Salman, A. Machine learning-powered mobile app for predicting used car prices. *Proceedings Of The 2021 3rd International Conference On Big-data Service And Intelligent Computation*. (2021)

[28] Valarmathi, B., Gupta, N., Santhi, K., Chellatamilan, T., Kavitha, A., Raahil, A. & Padmavathy, N. Price estimation of used cars using machine learning algorithms. *Lecture Notes Of The Institute For Computer Sciences, Social Informatics And Telecommunications Engineering*. pp. 26-41 (2023)

7 Appendix A

```
# R code for data cleaning
rm(list = ls())
library(tidyverse)
library(ggplot2)
library(dplyr)
library(data.table)
library(lubridate)
library(zoo)
library(xts)
library(quantmod)
library(rugarch)
library(rmgarch)



dir_in <- "./Data_in/"
dir_out <- "./Data_out/"

# Reading initial files
dt_in <-
  read.csv(
    paste0(dir_in, "CIS_Automotive_Kaggle_Sample.csv"),
    stringsAsFactors = F,
    check.names = F
  )

distinct_count <- dt_in %>% distinct(vin) %>% nrow()
print(distinct_count)

cols_keep <-
  c(
    "vin",
    "firstSeen",
    "lastSeen",
    "msrp",
    "askPrice",
    "mileage",
    "brandName",
    "modelName",
    "vf_BodyClass",
    "vf_ModelYear",
    "vf_EngineCylinders",
    "vf_TransmissionStyle",
    "vf_EngineHP",
    "vf_FuelTypePrimary",
    "vf_Doors",
```

```
    "vf_Seats"
  )

vehicle_types <- c(
  "Convertible/Cabriolet",
  "Sedan/Saloon",
  "Sport Utility Vehicle (SUV)/Multi-Purpose Vehicle (MPV)",
  "Pickup",
  "Wagon",
  "Hatchback/Liftback/Notchback",
  "Minivan",
  "Coupe",
  "Van",
  "Sport Utility Truck (SUT)"
)

# Define the date columns
date_cols <- c("firstSeen", "lastSeen")

# Define the columns to be converted to numeric
numeric_cols <-
  c(
    "msrp",
    "askPrice",
    "mileage",
    "vf_EngineHP",
    "vf_Doors",
    "vf_Seats",
    "vf_EngineCylinders",
    "vf_ModelYear"
  )

# Define the columns to be converted to character/string
character_cols <- c(
  "brandName",
  "modelName",
  "vf_BodyClass",
  "vf_FuelTypePrimary",
  "vf_TransmissionStyle"
)

# fixing different categorical variables classes
dt_in <- dt_in %>%
  select(all_of(cols_keep)) %>%
  mutate(across(everything(), str_trim)) %>%
  mutate(
    across(all_of(date_cols), as.Date, format = "%Y-%m-%d"),
```

```
  across(all_of(numeric_cols), as.numeric),
  across(all_of(character_cols), as.character),
  firstSeenYear = year(firstSeen)
) %>%
filter(
  askPrice > 100,
  askPrice < 3000000,
  vf_BodyClass %in% vehicle_types,
  !(mileage == 0 &
      vf_ModelYear < (firstSeenYear - 1)),
  modelName != "",
  vf_EngineHP > 60,
  vf_Seats < 10,
  mileage < 500000
) %>%
mutate(
  vf_TransmissionStyle = ifelse(
    vf_TransmissionStyle != "Manual/Standard",
    "Automatic",
    vf_TransmissionStyle
  ),
  vf_FuelTypePrimary = ifelse(
    vf_FuelTypePrimary == "Electric" &
      !is.na(vf_EngineCylinders),
    "Hybrid",
    vf_FuelTypePrimary
  ),
  vf_EngineCylinders = ifelse(
    vf_FuelTypePrimary == "Electric" &
      is.na(vf_EngineCylinders),
    0,
    vf_EngineCylinders
  ),
  vf_FuelTypePrimary = ifelse(!(
    vf_FuelTypePrimary %in% c("Electric", "Hybrid", "Gasoline", "Diesel")
  ), "Gasoline", vf_FuelTypePrimary),
  vf_BodyClass = ifelse(
    vf_BodyClass %in% c("Pickup", "Sport Utility Truck (SUT)"),
    "Truck",
    vf_BodyClass
  ),
  vf_BodyClass = ifelse(
    vf_BodyClass == "Sport Utility Vehicle (SUV)/Multi-Purpose Vehicle (MPV)",
    "SUV",
    vf_BodyClass
  ),
  vf_BodyClass = ifelse(
```

```
  vf_BodyClass == "Hatchback/Liftback/Notchback" ,
  "Hatchback",
  vf_BodyClass
),
vf_BodyClass = ifelse(vf_BodyClass == "Sedan/Saloon" , "Sedan", vf_BodyClass),
vf_BodyClass = ifelse(
  vf_BodyClass == "Convertible/Cabriolet" ,
  "Convertible",
  vf_BodyClass
),
vf_BodyClass = ifelse(vf_BodyClass == "Minivan" , "Van", vf_BodyClass),
vf_BodyClass = ifelse(vf_BodyClass == "Wagon", "SUV", vf_BodyClass),
vf_BodyClass = ifelse(
  modelName %in% c(
    "A4 allroad",
    "330i",
    "E-Class",
    "Golf Alltrack",
    "Golf SportWagen",
    "V60",
    "V90",
    "V90CC",
    "XC70"
  ) & vf_BodyClass == "SUV",
  "Wagon",
  vf_BodyClass
),
vf_BodyClass = ifelse(
  modelName %in% c("Transit", "Transit Connect",
                   "Sedona", "Promaster City") &
    vf_BodyClass == "SUV",
  "Van",
  vf_BodyClass
),
vf_BodyClass = ifelse(
  modelName == "Challenger" & vf_BodyClass == "Sedan",
  "Coupe",
  vf_BodyClass
),
vf_Doors = ifelse(vf_BodyClass == "Sedan", 4, vf_Doors),
vf_Doors = ifelse(vf_BodyClass == "Wagon", 5, vf_Doors),
vf_Doors = ifelse(is.na(vf_Doors) &
                    vf_BodyClass == "SUV", 5, vf_Doors),
vf_Doors = ifelse(is.na(vf_Doors) &
                    vf_BodyClass == "Hatchback", 5, vf_Doors),
vf_Doors = ifelse(vf_Doors == 6 &
                    vf_BodyClass == "Hatchback", 5, vf_Doors),
```

```
    vf_Doors = ifelse(is.na(vf_Doors) &
                        vf_BodyClass == "Van", 5, vf_Doors),
    vf_Doors = ifelse(is.na(vf_Doors) &
                        vf_BodyClass == "Truck", 4, vf_Doors),
    vf_Doors = ifelse(vf_BodyClass == "Convertible", 2, vf_Doors),
    vf_Doors = ifelse(is.na(vf_Doors) &
                        vf_BodyClass == "Coupe", 2, vf_Doors),
    vf_Seats = ifelse(vf_Doors %in% c(2, 3, 4, 5) &
                        is.na(vf_Seats), 5, vf_Seats),
    vf_Seats = ifelse(vf_Doors %in% c(4, 5) &
                        is.na(vf_Seats), 5, vf_Seats),
    age_at_sale = year(lastSeen) - vf_ModelYear,
    age_at_sale = ifelse(age_at_sale < 0, 0, age_at_sale)
  )

# fixing transmission names
dt_in$vf_TransmissionStyle <-
  gsub("Manual/Standard", "Manual", dt_in$vf_TransmissionStyle)

# getting rid of NAs
dt_in <- na.omit(dt_in)

# finding outliers
price_check <- dt_in %>%
  group_by(brandName, modelName, age_at_sale) %>%
  summarise(
    Q1 = quantile(askPrice, 0.25, na.rm = TRUE),
    Q3 = quantile(askPrice, 0.75, na.rm = TRUE),
    IQR = Q3 - Q1
  ) %>%
  ungroup() %>%
  mutate(IQR = ifelse(IQR < 0.1 * Q3, 0.1 * Q3, IQR),
         IQR = ifelse(IQR < 1000, 1000, IQR))

# filtering out outliers
dt_in2 <- dt_in %>%
  left_join(price_check, by = c("brandName", "modelName", "age_at_sale")) %>%
  filter(askPrice >= (Q1 - 1.5 * IQR) &
           askPrice <= (Q3 + 1.5 * IQR))

cols_keep2 <- c(
  "vin",
  "lastSeen",
  "askPrice",
  "mileage",
  "brandName",
  "modelName",
```

```
  "vf_BodyClass",
  "age_at_sale",
  "vf_EngineCylinders",
  "vf_TransmissionStyle",
  "vf_EngineHP",
  "vf_FuelTypePrimary",
  "vf_Doors",
  "vf_Seats"
)

# renaming the columns
final_dataset <- dt_in2 %>%
  select(all_of(cols_keep2)) %>%
  rename(
    Date = lastSeen,
    Price = askPrice,
    Mileage = mileage,
    Brand_Name = brandName,
    Model_Name = modelName,
    Body_Class = vf_BodyClass,
    Age_at_Sale = age_at_sale,
    Cylinders = vf_EngineCylinders,
    Transmission = vf_TransmissionStyle,
    Horse_Power = vf_EngineHP,
    Fuel_Type = vf_FuelTypePrimary,
    Doors = vf_Doors,
    Seats = vf_Seats
  )

# fixing car model names
final_dataset$Model_Name <- ifelse(
  final_dataset$Brand_Name == "BMW",
  gsub("^3.*", "3-Series", final_dataset$Model_Name),
  final_dataset$Model_Name
)

final_dataset$Model_Name <- ifelse(
  final_dataset$Brand_Name == "BMW",
  gsub("^2.*|^M240i", "2-Series", final_dataset$Model_Name),
  final_dataset$Model_Name
)

final_dataset$Model_Name <- ifelse(
  final_dataset$Brand_Name == "BMW",
  gsub("^4.*", "4-Series", final_dataset$Model_Name),
  final_dataset$Model_Name
)
```

```
final_dataset$Model_Name <- ifelse(
  final_dataset$Brand_Name == "BMW",
  gsub("^5.*|^M550i", "5-Series", final_dataset$Model_Name),
  final_dataset$Model_Name
)

final_dataset$Model_Name <- ifelse(
  final_dataset$Brand_Name == "BMW",
  gsub("^6.*", "6-Series", final_dataset$Model_Name),
  final_dataset$Model_Name
)

final_dataset$Model_Name <- ifelse(
  final_dataset$Brand_Name == "BMW",
  gsub("^7.*|^M760i", "7-Series", final_dataset$Model_Name),
  final_dataset$Model_Name
)

final_dataset$Model_Name <- ifelse(
  final_dataset$Brand_Name == "Lexus",
  gsub("LC500 / LC 500h", "LC", final_dataset$Model_Name),
  final_dataset$Model_Name
)

final_dataset$Model_Name <- ifelse(
  final_dataset$Brand_Name == "Lexus",
  gsub("LS/LS HYBRID", "LS", final_dataset$Model_Name),
  final_dataset$Model_Name
)

# saving final dataset
write.csv(final_dataset,
          paste0(dir_out, "CIS_auto_data_cleaned_01.csv"),
          row.names = FALSE)


# Python code for pre-processing, modeling, hyperparameter tuning and bootstrapping

import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf
from sklearn.preprocessing import OneHotEncoder, StandardScaler,
MinMaxScaler, PolynomialFeatures
```

```
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV, train_test_split,
KFold, cross_val_score, TimeSeriesSplit, cross_validate
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.metrics import r2_score, mean_squared_error,
accuracy_score, mean_absolute_error, make_scorer,
mean_absolute_percentage_error, PredictionErrorDisplay
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import GradientBoostingRegressor, HistGradientBoostingRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.svm import SVR, LinearSVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.cross_decomposition import PLSRegression
from sklearn.tree import DecisionTreeRegressor
import missingno as msno
from sklearn.utils import shuffle
import warnings
import time
np.set_printoptions(precision = 5, suppress = True)
warnings.filterwarnings("ignore")
sns.set(rc = {
  'figure.figsize':(20, 20)
})
%matplotlib inline

def get_one_hot_feature_names(encoder, input_features):
# Extracting categories identified by the encoder
  encoder_categories = encoder.categories_

# Generating new feature names
new_feature_names = []
for feature, categories in zip(input_features, encoder_categories):
  new_feature_names.extend([f"{feature}_{category}" for category in categories])

return new_feature_names

def add_model_results(brand,
                      model_name,
                      y_pred_test,
                      y_pred_train,
                      y_test,
                      y_train,
                      results_df,
                      total_time):mse_train = mean_squared_error(y_train, y_pred_train)
mae_train = mean_absolute_error(y_train, y_pred_train)
r2_train = r2_score(y_train, y_pred_train)
mape_train = mean_absolute_percentage_error(y_train, y_pred_train)
```

```
mse_test = mean_squared_error(y_test, y_pred_test)
mae_test = mean_absolute_error(y_test, y_pred_test)
r2_test = r2_score(y_test, y_pred_test)
mape_test = mean_absolute_percentage_error(y_test, y_pred_test)

time = f'{total_time:.2f} s'

new_record = pd.DataFrame({
  'Brand Name':[brand],
  'Model':[model_name],
  'MSE Train':[mse_train],
  'MAE Train':[mae_train],
  'R^2 Train':[r2_train],
  'MAPE Train':[mape_train],
  'MSE Test':[mse_test],
  'MAE Test':[mae_test],
  'R^2 Test':[r2_test],
  'MAPE Test':[mape_test],
  'Time':[time]
})
return pd.concat([results_df, new_record], ignore_index = True)

data_in = pd.read_csv('./Data_out/CIS_auto_data_cleaned_01.csv')

data_in['Date'] = pd.to_datetime(data_in['Date'])

data_in['Month'] = data_in['Date'].dt.month

# change model name
data = data_in[data_in['Brand_Name'].isin(['FORD'])].reset_index()

model_counts = data.groupby('Model_Name').size()

y = data[['Price']]
X = data[['Month',
          'Mileage',
          'Model_Name',
          'Body_Class',
          'Age_at_Sale',
          'Horse_Power',
          'Fuel_Type',
          'Doors',
          'Seats']]

# train-test split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 100)

# categorical variables encoding
tr_cols = ['Month',
           'Model_Name',
           'Body_Class',
           'Fuel_Type']
encoder = OneHotEncoder()
encoder.fit(X_test[tr_cols])

column_names = get_one_hot_feature_names(encoder, tr_cols)

# Capture the index before the transformation
train_index = X_train.index
test_index = X_test.index


encoded_train = encoder.transform(X_train[tr_cols])
encoded_test = encoder.transform(X_test[tr_cols])


encoded_train_df = pd.DataFrame(encoded_train.toarray(),
                                index = train_index,
                                columns = column_names)
encoded_test_df = pd.DataFrame(encoded_test.toarray(), index = test_index, columns =
                                column_names)

num_cols = ['Mileage',
            'Age_at_Sale',
            'Horse_Power',
            'Doors',
            'Seats']


# df of numeric vars
X_train_num_df = X_train[num_cols]
X_test_num_df = X_test[num_cols]

# numpy array of numeric vars for scikit learn use
X_train_num = X_train_num_df.to_numpy()
X_test_num = X_test_num_df.to_numpy()

## Min Max scaled numeric variables with default feature_range=(0, 1)
scaler = MinMaxScaler()
scaler.fit(X_train_num_df)
X_train_scaled = scaler.transform(X_train_num_df)
X_test_scaled = scaler.transform(X_test_num_df)
```

```
# Get the index and column names from the original DataFrames
train_index = X_train_num_df.index
test_index = X_test_num_df.index
columns = X_train_num_df.columns

# Convert the scaled NumPy arrays back to DataFrames
X_train_scaled_df = pd.DataFrame(X_train_scaled, index = train_index, columns =
                                 columns)
X_test_scaled_df = pd.DataFrame(X_test_scaled, index = test_index, columns =
                                columns)

# Not scaled full df
X_train_gen_full_df = pd.concat([X_train_num_df, encoded_train_df], axis =
                                1)
X_test_gen_full_df = pd.concat([X_test_num_df, encoded_test_df], axis =
                               1)

X_train_gen_full = X_train_gen_full_df.to_numpy()
X_test_gen_full = X_test_gen_full_df.to_numpy()


# Adding encoded categorical variables to scaled numerical ones
X_train_scaled_full_df = pd.concat([X_train_scaled_df, encoded_train_df], axis =
                                   1)
X_test_scaled_full_df = pd.concat([X_test_scaled_df, encoded_test_df], axis =
                                  1)

X_train_scaled_full = X_train_scaled_full_df.to_numpy()
X_test_scaled_full = X_test_scaled_full_df.to_numpy()

# Ford Linear regression not scaled
model = LinearRegression()
model.fit(X_train_gen_full, y_train)

mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_gen_full_df))
y_pred = model.predict(X_test_gen_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
```

```
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("OLS Model - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# Ford Linear regression scaled
model = LinearRegression()
model.fit(X_train_scaled_full, y_train)

mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("OLS Model - Ford - Scaled Numerical Variables")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# Ford Linear SVR Regression scaled

start_time = time.time()

model = LinearSVR()
model.fit(X_train_scaled_full, y_train)
```

```
end_time = time.time()

mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("Linear SVR Model - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# Ford Gaussian SVR Regression scaled

start_time = time.time()

model = SVR(kernel = "rbf")
model.fit(X_train_scaled_full, y_train)

end_time = time.time()

mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}
```

```
total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("Gaussian Kernel SVR Model - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# Support Vector Regression
start_time = time.time()

model = LinearSVR(dual = "auto")

# Define the parameter grid
param_grid = {
  'epsilon':[0, 0.1, 0.2, 0.5, 1],
  'C':[0.1, 1, 10, 100, 1000]
}

# Create a scorer
mape_scorer = make_scorer(mean_absolute_percentage_error, greater_is_better =
                          False)

# Create the grid search object
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
  scoring = mape_scorer,
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())
# Note: y_train should be a 1D array
```

```
# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)

model = LinearSVR(C = best_params['C'],
                  epsilon = best_params['epsilon'],
                  dual = "auto")
model.fit(X_train_scaled_full, y_train)
end_time = time.time()

mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("Linear SVR Model - Epsilon: 1, C: 100 - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# k-neighbors regression

start_time = time.time()

model = KNeighborsRegressor()
model.fit(X_train_scaled_full, y_train)

end_time = time.time()
```

```
mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("K-Neighbors Model - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# k-neighbors regression
start_time = time.time()
model = KNeighborsRegressor()

# Define the parameter grid
param_grid = {
  'n_neighbors':[3, 5, 7, 10, 15],
  'weights':['uniform', 'distance'],
  'metric':['minkowski', 'euclidean', 'manhattan']
}

# Create a scorer
mape_scorer = make_scorer(mean_squared_error, greater_is_better = False)

# Create the grid search object
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
```

```
  scoring = mape_scorer,
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())
# Note: y_train should be a 1D array

# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)

model = KNeighborsRegressor(n_neighbors = best_params['n_neighbors'],
                            weights = best_params['weights'],
                            metric = best_params['metric'])
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("K-Neighbors - 15, distance, manhattan - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# Decision Trees Regression
```

```
start_time = time.time()
model = DecisionTreeRegressor()
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("Decision Trees Regression Model - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# Decision Trees Regression
start_time = time.time()
model = DecisionTreeRegressor()

# Define the parameter grid
param_grid = {
  'max_depth':[None, 5, 10, 20, 30],  # Assuming max depth varies from 1 to 20
  'min_samples_split':[2, 5, 10, 20],  # Assuming min_samples_split varies from 2 to 20
  'min_samples_leaf':[1, 2, 5, 10],  # Assuming min_samples_leaf varies from 1 to 20
}

# Create a scorer
mape_scorer = make_scorer(mean_absolute_percentage_error, greater_is_better =
                          False)

# Create the grid search object
```

```
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
  scoring = mape_scorer,
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())

# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)

# Fit the model with the optimized parameters
model = DecisionTreeRegressor(
  max_depth = best_params['max_depth'],
  min_samples_split = best_params['min_samples_split'],
  min_samples_leaf = best_params['min_samples_leaf'],
)

model.fit(X_train_scaled_full, y_train)
end_time = time.time()
mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
```

```
)
ax.set_title("Decision Trees - 'max_depth': 30,
'min_samples_leaf': 2,
'min_samples_split': 5 - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# Gradient Boosting Regression
start_time = time.time()
model = GradientBoostingRegressor()
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("Gradient Boosting Regression Model - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# Gradient Boosting Regression
start_time = time.time()
model = GradientBoostingRegressor()

# Define the parameter grid
param_grid = {
  'n_estimators':[100, 200, 300],
```

```
  'learning_rate':[0.01, 0.1, 0.2],
  'max_depth':[3, 5, 10, None]
}

# Create a scorer
mape_scorer = make_scorer(mean_absolute_percentage_error, greater_is_better =
                          False)

# Create the grid search object
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
  scoring = mape_scorer,
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())

# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)

# Fit the model with the optimized parameters
model = GradientBoostingRegressor(
  n_estimators = best_params['n_estimators'],
  learning_rate = best_params['learning_rate'],
  max_depth = best_params['max_depth']
)

model.fit(X_train_scaled_full, y_train)
end_time = time.time()
mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
```

```
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("Gradient Boosting - 'learning_rate': 0.1,
'max_depth': 10,
'n_estimators': 300 - Ford")

for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# MLP Regression
start_time = time.time()
model = model = MLPRegressor()
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("MLP Regression Model - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
```

```
ax.legend(loc = "upper left")
plt.tight_layout()

# MLP Regression
start_time = time.time()
model = MLPRegressor()

# Define the parameter grid
# Define the parameter grid
param_grid = {
  'hidden_layer_sizes':[(60, 60), (80, 80), (100, 100)],
  'alpha':[0.1, 0.15, 0.2],
  'learning_rate_init':[0.1],
  'max_iter':[400],
  'early_stopping':[True],
  'validation_fraction':[0.1]
}

# Create a scorer
mape_scorer = make_scorer(mean_absolute_percentage_error, greater_is_better =
                          False)

# Create the grid search object
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
  scoring = mape_scorer,
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())

# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)


# Fit the model with the optimized parameters
model = MLPRegressor(
  hidden_layer_sizes = best_params['hidden_layer_sizes'],
  alpha = best_params['alpha'],
  learning_rate_init = best_params['learning_rate_init'],
  # Use the best learning rate
```

```
  max_iter = 400,
  # Increased number of iterations
  early_stopping = True,
  # Enable early stopping
  validation_fraction = 0.1  # Set aside 10% of data for validation
)

model.fit(X_train_scaled_full, y_train)
end_time = time.time()
mape_train = mean_absolute_percentage_error(y_train, model.predict(X_train_scaled_full_df))
y_pred = model.predict(X_test_scaled_full_df)
mape_test = mean_absolute_percentage_error(y_test, y_pred)
scores = {
  "MAPE on training set":f"{mape_train:.2f}",
  "MAPE on testing set":f"{mape_test:.2f}",
}

total_time = end_time - start_time
print(f"Total time taken: {total_time:.2f} seconds")

# Plotting results
_, ax = plt.subplots(figsize = (5, 5))
display = PredictionErrorDisplay.from_predictions(
  y_test,
  y_pred,
  kind = "actual_vs_predicted",
  ax = ax,
  scatter_kwargs = {
    "alpha":0.5
  }
)
ax.set_title("MLP - 'alpha': 0.2, 'hidden_layer_sizes': (80, 80) - Ford")
for name, score in scores.items():ax.plot([], [], " ", label = f"{name}: {score}")
ax.legend(loc = "upper left")
plt.tight_layout()

# Get all unique brand names
# Get counts of each brand
brand_counts = data_in['Brand_Name'].value_counts()

# Get the top 3 brands
top_brands = brand_counts.head(3).index

# Initialize a DataFrame to store the results
model_results = pd.DataFrame(columns = ['Brand Name',
'Model',
'MSE Train',
```

```
'MAE Train',
'R^2 Train',
'MAPE Train',
'MSE Test',
'MAE Test',
'R^2 Test',
'MAPE Test',
'Time'])

# Loop through each brand name
for brand in top_brands:# Filter data for the current brand
  brand_data = data_in[data_in['Brand_Name'] == brand]

# Display or further process data for the current brand
print(f"Data for brand: {brand}")

# Data pre-processing
y = brand_data[['Price']]
X = brand_data[['Month',
                'Mileage',
                'Model_Name',
                'Body_Class',
                'Age_at_Sale',
                'Horse_Power',
                'Fuel_Type',
                'Doors',
                'Seats']]

# train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 100)

# columns to encode
tr_cols = ['Month',
           'Model_Name',
           'Body_Class',
           'Fuel_Type']

encoder = OneHotEncoder()
encoder.fit(X_test[tr_cols])

column_names = get_one_hot_feature_names(encoder, tr_cols)

# Capture the index before the transformation
train_index = X_train.index
test_index = X_test.index

# encoding
```

```
encoded_train = encoder.transform(X_train[tr_cols])
encoded_test = encoder.transform(X_test[tr_cols])


encoded_train_df = pd.DataFrame(encoded_train.toarray(),
                                index = train_index,
                                columns = column_names)
encoded_test_df = pd.DataFrame(encoded_test.toarray(), index = test_index, columns =
                                 column_names)


num_cols = ['Mileage',
            'Age_at_Sale',
            'Horse_Power',
            'Doors',
            'Seats']

# df of numeric vars
X_train_num_df = X_train[num_cols]
X_test_num_df = X_test[num_cols]

## Min Max scaled numeric variables with default feature_range=(0, 1)
scaler = MinMaxScaler()
scaler.fit(X_train_num_df)
X_train_scaled = scaler.transform(X_train_num_df)
X_test_scaled = scaler.transform(X_test_num_df)


# Get the index and column names from the original DataFrames
train_index = X_train_num_df.index
test_index = X_test_num_df.index
columns = X_train_num_df.columns

# Convert the scaled NumPy arrays back to DataFrames
X_train_scaled_df = pd.DataFrame(X_train_scaled, index = train_index, columns =
                                   columns)
X_test_scaled_df = pd.DataFrame(X_test_scaled, index = test_index, columns =
                                  columns)

# Adding encoded categorical variables to transformed numerical ones
X_train_scaled_full_df = pd.concat([X_train_scaled_df, encoded_train_df], axis =
                                     1)
X_test_scaled_full_df = pd.concat([X_test_scaled_df, encoded_test_df], axis =
                                    1)

X_train_scaled_full = X_train_scaled_full_df.to_numpy()
X_test_scaled_full = X_test_scaled_full_df.to_numpy()
```

```
# Linear regression ##############################################################################
print('Modeling Linear Regression')

start_time = time.time()
model = LinearRegression()
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results = add_model_results(
  brand,
  'Linear Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results,
  total_time
)


# Support Vector Regression ######################################################

print('Modeling Support Vector Regression')

start_time = time.time()
model = LinearSVR(dual = "auto")

# Define the parameter grid
param_grid = {
  'epsilon':[0, 0.1, 0.2, 0.5, 1],
  'C':[0.1, 1, 10, 100, 1000]
}

# Create a scorer
mape_scorer = make_scorer(mean_absolute_percentage_error, greater_is_better =
                          False)

# Create the grid search object
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
  scoring = mape_scorer,
```

```
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())  # Note: y_train should be a 1D array

# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)

model = LinearSVR(C = best_params['C'],
                  epsilon = best_params['epsilon'],
                  dual = "auto")
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results = add_model_results(
  brand,
  'Support Vector Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results,
  total_time
)

# K-Neighbors regression ##########################################

print('Modeling K-Neighbors Regression')

start_time = time.time()
model = KNeighborsRegressor()

# Define the parameter grid
param_grid = {
  'n_neighbors':[3, 5, 7, 10, 15],
  'weights':['uniform', 'distance'],
  'metric':['minkowski', 'euclidean', 'manhattan']
}

# Create a scorer
```

```
mape_scorer = make_scorer(mean_absolute_percentage_error, greater_is_better =
                          False)

# Create the grid search object
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
  scoring = mape_scorer,
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())  # Note: y_train should be a 1D array

# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)

model = KNeighborsRegressor(n_neighbors = best_params['n_neighbors'],
                            weights = best_params['weights'],
                            metric = best_params['metric'])
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time


model_results = add_model_results(
  brand,
  'K-Neighbors Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results,
  total_time
)

# Decisison Trees Regression ###############################################

print('Modeling Decisison Trees Regression')

start_time = time.time()
```

```
model = DecisionTreeRegressor()

# Define the parameter grid
param_grid = {
  'max_depth':[None, 5, 10, 20, 30],  # Assuming max depth varies from 1 to 20
  'min_samples_split':[2, 5, 10, 20],  # Assuming min_samples_split varies from 2 to 20
  'min_samples_leaf':[1, 2, 5, 10],  # Assuming min_samples_leaf varies from 1 to 20
}

# Create a scorer
mape_scorer = make_scorer(mean_absolute_percentage_error, greater_is_better =
                          False)

# Create the grid search object
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
  scoring = mape_scorer,
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())

# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)

# Fit the model with the optimized parameters
model = DecisionTreeRegressor(
  max_depth = best_params['max_depth'],
  min_samples_split = best_params['min_samples_split'],
  min_samples_leaf = best_params['min_samples_leaf']
)

model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results = add_model_results(
  brand,
  'Decisison Trees Regression',
```

```
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results,
  total_time
)

# Gradient Boosting Regression ###################################################

print('Modeling Gradient Boosting Regression')

start_time = time.time()
model = GradientBoostingRegressor()

# Define the parameter grid
param_grid = {
  'n_estimators':[100, 200, 300],
  'learning_rate':[0.01, 0.1, 0.2],
  'max_depth':[3, 5, 10, None]
}

# Create a scorer
mape_scorer = make_scorer(mean_absolute_percentage_error, greater_is_better =
                          False)

# Create the grid search object
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
  scoring = mape_scorer,
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())

# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)

# Fit the model with the optimized parameters
model = GradientBoostingRegressor(
  n_estimators = best_params['n_estimators'],
```

```
  learning_rate = best_params['learning_rate'],
  max_depth = best_params['max_depth']
)
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results = add_model_results(
  brand,
  'Gradient Boosting Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results,
  total_time
)

# MLP Regression ################################################

print('Modeling MLP Regression')

start_time = time.time()
model = MLPRegressor()

# Define the parameter grid
param_grid = {
  'hidden_layer_sizes':[(60, 60), (80, 80), (100, 100)],
  'alpha':[0.1, 0.15, 0.2],
  'learning_rate_init':[0.1],
  'max_iter':[400],
  'early_stopping':[True],
  'validation_fraction':[0.1]
}

# Create a scorer
mape_scorer = make_scorer(mean_absolute_percentage_error, greater_is_better =
                          False)

# Create the grid search object
grid_search = GridSearchCV(
  model,
  param_grid,
  cv = 5,
  scoring = mape_scorer,
```

```
  verbose = 1,
  n_jobs = -1
)

# Fit the model to the training data
grid_search.fit(X_train_scaled_full, y_train.values.ravel())

# Get the best parameters from the grid search
best_params = grid_search.best_params_

print(best_params)

# Fit the model with the optimized parameters
model = MLPRegressor(
  hidden_layer_sizes = best_params['hidden_layer_sizes'],
  alpha = best_params['alpha'],
  learning_rate_init = 0.1,
  max_iter = 400,
  # Increased number of iterations
  early_stopping = True,
  # Enable early stopping
  validation_fraction = 0.1  # Set aside 10% of data for validation
)

model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results = add_model_results(
  brand,
  'MLP Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results,
  total_time
)


model_results

# Default models
# Get all unique brand names
# Get counts of each brand
```

```
brand_counts = data_in['Brand_Name'].value_counts()

# Get the top 3 brands
top_brands = brand_counts.head(3).index

# Initialize a DataFrame to store the results
model_results2 = pd.DataFrame(columns = ['Brand Name',
                                         'Model',
                                         'MSE Train',
                                         'MAE Train',
                                         'R^2 Train',
                                         'MAPE Train',
                                         'MSE Test',
                                         'MAE Test',
                                         'R^2 Test',
                                         'MAPE Test',
                                         'Time'])


# Loop through each brand name
for brand in top_brands:# Filter data for the current brand
  brand_data = data_in[data_in['Brand_Name'] == brand]

# Display or further process data for the current brand
print(f"Data for brand: {brand}")

# data pre-processing
y = brand_data[['Price']]
X = brand_data[['Month',
                'Mileage',
                'Model_Name',
                'Body_Class',
                'Age_at_Sale',
                'Horse_Power',
                'Fuel_Type',
                'Doors',
                'Seats']]

# train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 100)

# columns to encode
tr_cols = ['Month',
           'Model_Name',
           'Body_Class',
           'Fuel_Type']
```

```
encoder = OneHotEncoder()
encoder.fit(X_test[tr_cols])

column_names = get_one_hot_feature_names(encoder, tr_cols)

# Capture the index before the transformation
train_index = X_train.index
test_index = X_test.index

# encoding
encoded_train = encoder.transform(X_train[tr_cols])
encoded_test = encoder.transform(X_test[tr_cols])


encoded_train_df = pd.DataFrame(encoded_train.toarray(),
                                index = train_index,
                                columns = column_names)
encoded_test_df = pd.DataFrame(encoded_test.toarray(), index = test_index, columns =
                                 column_names)


num_cols = ['Mileage',
            'Age_at_Sale',
            'Horse_Power',
            'Doors',
            'Seats']

# df of numeric vars
X_train_num_df = X_train[num_cols]
X_test_num_df = X_test[num_cols]

## Min Max scaled numeric variables with default feature_range=(0, 1)
scaler = MinMaxScaler()
scaler.fit(X_train_num_df)
X_train_scaled = scaler.transform(X_train_num_df)
X_test_scaled = scaler.transform(X_test_num_df)


# Get the index and column names from the original DataFrames
train_index = X_train_num_df.index
test_index = X_test_num_df.index
columns = X_train_num_df.columns

# Convert the scaled NumPy arrays back to DataFrames
X_train_scaled_df = pd.DataFrame(X_train_scaled, index = train_index, columns =
                                   columns)
X_test_scaled_df = pd.DataFrame(X_test_scaled, index = test_index, columns =
```

```
                                    columns)

# Adding encoded categorical variables to transformed numerical ones
X_train_scaled_full_df = pd.concat([X_train_scaled_df, encoded_train_df], axis =
                                     1)
X_test_scaled_full_df = pd.concat([X_test_scaled_df, encoded_test_df], axis =
                                    1)

X_train_scaled_full = X_train_scaled_full_df.to_numpy()
X_test_scaled_full = X_test_scaled_full_df.to_numpy()

# Linear regression ###########################################################
print('Modeling Linear Regression')

start_time = time.time()
model = LinearRegression()
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results2 = add_model_results(
  brand,
  'Linear Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results2,
  total_time
)

# Support Vector Regression ###################################################

print('Modeling Support Vector Regression')

start_time = time.time()
model = LinearSVR(dual = "auto")
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results2 = add_model_results(
  brand,
```

```
  'Support Vector Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results2,
  total_time
)

# K-Neighbors regression ##############################################################################

print('Modeling K-Neighbors Regression')

start_time = time.time()
model = KNeighborsRegressor()
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results2 = add_model_results(
  brand,
  'K-Neighbors Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results2,
  total_time
)

# Decisison Trees Regression ##########################################################

print('Modeling Decisison Trees Regression')

start_time = time.time()
model = DecisionTreeRegressor()

model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results2 = add_model_results(
  brand,
```

```
  'Decisison Trees Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results2,
  total_time
)

# Gradient Boosting Regression ################################################################

print('Modeling Gradient Boosting Regression')

start_time = time.time()
model = GradientBoostingRegressor()
model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results2 = add_model_results(
  brand,
  'Gradient Boosting Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results2,
  total_time
)

# MLP Regression ##############################################################

print('Modeling MLP Regression')

start_time = time.time()
model = MLPRegressor()

model.fit(X_train_scaled_full, y_train)
end_time = time.time()
y_pred_train = model.predict(X_train_scaled_full)
y_pred_test = model.predict(X_test_scaled_full)
total_time = end_time - start_time

model_results2 = add_model_results(
  brand,
```

```
  'MLP Regression',
  y_pred_test,
  y_pred_train,
  y_test,
  y_train,
  model_results2,
  total_time
)


model_results2

model_results_tuned = model_results.copy()
model_results_tuned = model_results_tuned.drop(['Time',
                                                'MSE Train',
                                                'MAE Train',
                                                'MAPE Train',
                                                'R^2 Train',
                                                'R^2 Test'], axis = 1)
model_results_default = model_results2.copy()
model_results_default = model_results_default.drop(['MSE Train',
                                                    'MAE Train',
                                                    'MAPE Train',
                                                    'R^2 Train',
                                                    'R^2 Test'],  axis = 1)
model_results_default['Time'] = model_results_default['Time'].str.replace(' seconds', ' s')
model_results_default['MSE Test'] = model_results_default['MSE Test'].round(1)
model_results_default['MAE Test'] = model_results_default['MAE Test'].round(1)
model_results_tuned['MSE Test'] = model_results_tuned['MSE Test'].round(1)
model_results_tuned['MAE Test'] = model_results_tuned['MAE Test'].round(1)


model_results_default

def dataframe_to_latex(df, title, label):# Convert DataFrame to LaTeX
  latex_str = df.to_latex(
    index = False,
    column_format = '|'+'l|' * len(df.columns),
    caption = title,
    label = label,
    position = 'h!',
    longtable = False
  )

return latex_str
```

```
latex_table = dataframe_to_latex(model_results_default,
                                 "Models Comparison Results",
                                 "tab:model-comparison")
print(latex_table)

latex_table = dataframe_to_latex(
  model_results_tuned,
  "Tuned Models Comparison Results",
  "tab:model-tuned-comparison"
)
print(latex_table)

# Averaging predictions
def average_predictions(models, X):predictions = np.array([model.predict(X) for model in models])
return np.mean(predictions, axis = 0)

# code for bootstrapping
from sklearn.utils import resample

brand_counts = data_in['Brand_Name'].value_counts()

# Get the top 3 brands
top_brands = brand_counts.head(3).index

# Initialize a DataFrame to store the results
model_results_boot = pd.DataFrame(columns = ['Brand Name', 'MSE Test', 'MAE Test', 'Train Price Varia

# Initialize a DataFrame to store the results for each model
model_results_per_model = pd.DataFrame(columns = ['Brand Name', 'Car Model', 'Train Price Variance',

# Loop through each brand name
for brand in top_brands:# Filter data for the current brand
  brand_data = data_in[data_in['Brand_Name'] == brand]

unique_models = brand_data['Model_Name'].unique()

# Display or further process data for the current brand
print(f"Data for brand: {brand}")

# data pre-processing
y = brand_data[['Price']]
X = brand_data[['Month',
                'Mileage',
                'Model_Name',
                'Body_Class',
                'Age_at_Sale',
                'Horse_Power',
```

```
                'Fuel_Type',
                'Doors',
                'Seats']]

# train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 100)

tr_cols = ['Month',
           'Model_Name',
           'Body_Class',
           'Fuel_Type']

encoder = OneHotEncoder()
encoder.fit(X_test[tr_cols])

column_names = get_one_hot_feature_names(encoder, tr_cols)

# Capture the index before the transformation
train_index = X_train.index
test_index = X_test.index


encoded_train = encoder.transform(X_train[tr_cols])
encoded_test = encoder.transform(X_test[tr_cols])


encoded_train_df = pd.DataFrame(encoded_train.toarray(),
                                index = train_index,
                                columns = column_names)
encoded_test_df = pd.DataFrame(encoded_test.toarray(), index = test_index, columns =
                                column_names)


num_cols = ['Mileage',
            'Age_at_Sale',
            'Horse_Power',
            'Doors',
            'Seats']

# df of numeric vars
X_train_num_df = X_train[num_cols]
X_test_num_df = X_test[num_cols]

## Min Max scaled numeric variables with default feature_range=(0, 1)
scaler = MinMaxScaler()
scaler.fit(X_train_num_df)
X_train_scaled = scaler.transform(X_train_num_df)
```

```
X_test_scaled = scaler.transform(X_test_num_df)


# Get the index and column names from the original DataFrames
train_index = X_train_num_df.index
test_index = X_test_num_df.index
columns = X_train_num_df.columns

# Convert the scaled NumPy arrays back to DataFrames
X_train_scaled_df = pd.DataFrame(X_train_scaled, index = train_index, columns =
                                 columns)
X_test_scaled_df = pd.DataFrame(X_test_scaled, index = test_index, columns =
                                columns)

# Adding encoded categorical variables to transformed numerical ones
X_train_scaled_full_df = pd.concat([X_train_scaled_df, encoded_train_df], axis =
                                   1)
X_test_scaled_full_df = pd.concat([X_test_scaled_df, encoded_test_df], axis =
                                  1)

X_train_scaled_full = X_train_scaled_full_df.to_numpy()
X_test_scaled_full = X_test_scaled_full_df.to_numpy()

# Number of bootstrap samples
n_bootstraps = 1000

# Create bootstrapped models
models = []
for _ in range(n_bootstraps):X_sample, y_sample = resample(X_train_scaled_full, y_train)
model = DecisionTreeRegressor(
  max_depth = None,
  min_samples_leaf = 1,
  min_samples_split = 2
)
model.fit(X_sample, y_sample)
models.append(model)

# Predict and evaluate
y_pred_avg = average_predictions(models, X_test_scaled_full)
mse_test = mean_squared_error(y_test, y_pred_avg)
mae_test = mean_absolute_error(y_test, y_pred_avg)
mape_test = mean_absolute_percentage_error(y_test, y_pred_avg)
r2_test = r2_score(y_test, y_pred_avg)
#Variance
test_var_brand = y_test['Price'].std()
train_var_brand = y_train['Price'].std()
```

```
# Count of observations
count_observations = len(X_train_scaled_full_df)

new_record = pd.DataFrame({
  'Brand Name':[brand],
  'MSE Test':[mse_test],
  'MAE Test':[mae_test],
  'Train Price Variance':[train_var_brand],
  'Test Price Variance':[test_var_brand],
  'R^2 Test':[r2_test],
  'MAPE Test':[mape_test],
  'Observations Trained':[count_observations]
})
model_results_boot = pd.concat([model_results_boot, new_record], ignore_index =
                                True)

X_test_scaled_full_df = X_test_scaled_full_df.reset_index(drop = True)
X_train_scaled_full_df = X_test_scaled_full_df.reset_index(drop = True)
y_test = y_test.reset_index(drop = True)
y_train = y_train.reset_index(drop = True)


for model_name in unique_models:# Filter the test set for the current model
  print(f"Data for model: {model_name}")
test_indices = X_test_scaled_full_df[X_test_scaled_full_df['Model_Name_'+model_name] == 1].index
train_indices = X_train_scaled_full_df[X_train_scaled_full_df['Model_Name_'+model_name] == 1].index
y_test_model = y_test.loc[test_indices]
y_train_model = y_train.loc[test_indices]
X_test_model = X_test_scaled_full[test_indices]

# Average predictions for the current model
y_pred_avg_model = y_pred_avg[test_indices]

# Calculate metrics
mape_test = mean_absolute_percentage_error(y_test_model, y_pred_avg_model)

# Count of observations
count_observations_model = len(train_indices)

#Variance
test_var_model = y_test_model['Price'].var()
train_var_model = y_train_model['Price'].var()

r2_test_model = r2_score(y_test_model, y_pred_avg_model)

# Store results
new_record2 = pd.DataFrame({
```

```
  'Brand Name':[brand],
  'Car Model':[model_name],
  'Train Price Variance':[train_var_model],
  'Test Price Variance':[test_var_model],
  'R^2 Test':[r2_test_model],
  'MAPE Test':[mape_test],
  'Observations Trained':[count_observations_model]
})


model_results_per_model = pd.concat([model_results_per_model, new_record2], ignore_index =
                                    True)

model_results_per_model

model_results_boot

latex_table = dataframe_to_latex(
  model_results_boot,
  "Bootstrapped Decision Trees Regression Results",
  "tab:bootstrap-comparison"
)
print(latex_table)

# estimating Decision Trees Regression to evaluate feature importance
model = DecisionTreeRegressor(
  max_depth = None,
  min_samples_leaf = 1,
  min_samples_split = 2
)
model.fit(X_train_scaled_full, y_train)
feature_importances = model.feature_importances_

# To display feature importances along with their corresponding feature names
feature_names = X_train_scaled_full_df.columns  # Replace with your actual feature names
# Create a DataFrame and sort by importance
importances = pd.DataFrame({
  'Feature':feature_names, 'Importance':feature_importances
})
importances_sorted = importances.sort_values(by = 'Importance', ascending =
                                             False).head(10)

print(importances_sorted)

latex_table = dataframe_to_latex(
  importances_sorted,
  "Decision Trees Regression - Feature Importance - Ford",
```

```
  "tab:bootstrap-comparison"
)
print(latex_table)
```