Machine Learning Project Report

January 29, 2025

Group Members

- Lingfeng Chen,
- Asier Aurre,
- Giovanni Pacchetti
- Martín Fernández de Retana

Dataset

- **Dataset Name:** Cars Specifications Dataset
- **URL:** Cars-dataset

Dataset Description

The Cars Specifications Dataset contains extensive details about various second hand car models, including technical, performance, environmental, and feature-based information. The dataset covers a variety of feature from cars with vastly different data types and ranges. Due to this, we believe it is suitable for tasks like predictive modeling (e.g., fuel consumption, CO2 emissions), classification, and clustering based on car features.

Note: The data was collected through web scraping techniques.

Tasks to Solve

In this project, we aim to solve the following machine learning tasks using the Cars Specifications Dataset:

- **Regression Task:** Predict the car's price based on its technical and environmental specifications, such as mileage, engine capacity, fuel type, CO2 emissions, and trunk capacity.
- **Classification Task:** Classify cars into different environmental labels (e.g., ECO, C, D) based on features like fuel type, emissions, and consumption.
- **Clustering Task:** Cluster cars based on their technical features (e.g., engine power, transmission type, body type, brand) to identify distinct groups of cars with similar attributes.
- **Algorithms comparison:** Compare classic machine learning algorithms against deep learning algorithms (e.g., ANN vs multi-variable regression)

Features/Variables of the Dataset

The dataset consists of a variety of features organized into several categories, such as general, interior or exterior finish or electrical features. Below is a categorized list of some of the key variables in the dataset, among many others.

Feature	Description
price	The price of the car in local currency (e.g., EUR).
km	The mileage of the car in kilometers.
year	The manufacturing year of the car.
color	The color of the car.
cubicCapacity	The engine capacity in cubic centimeters (cc).
brand	The brand or manufacturer of the car (e.g., Mazda, BMW).
model	The specific model of the car.
fuelType	The type of fuel the car uses (e.g., Gasoline, Diesel).
seatingCapacity	The seating capacity of the car.
warranty_months	The warranty duration in months.
province	The province where the car is being sold.
environmentalLabel	The environmental classification of the car (e.g., C, ECO).
maxSpeed	The maximum speed of the car in km/h.
acceleration	The time it takes to accelerate from 0 to 100 km/h in seconds.
power_cv	Engine power in horsepower (cv).
power_kw	Engine power in kilowatts (kW).
rpm_max_power	The engine's revolutions per minute at maximum power.
max_torque_nm	The maximum torque of the engine in Newton-meters.
rpm_max_torque	The engine's revolutions per minute at maximum torque.
transmission_description	Detailed description of the car's transmission system.
brakes	Description of the car's braking system.
traction	The type of traction (e.g., Front-wheel, Rear-wheel, All-wheel).
dimensions.width	The width of the car in millimeters.
dimensions.height	The height of the car in millimeters.
dimensions.length	The length of the car in millimeters.

Note: There are many more features in the dataset related to comfort, entertainment, safety, and electric vehicle charging, which are not listed above for brevity since there are 146 columns in total.

Bonus Task

- **Natural Language Processing:** Predict the car's price based on its descriptions and the mileage, applying techniques such as embeddings and artificial neural networks (ANN).
- **Computer Vision:** Using pictures as parameters to classify the car's brand using an Conventional Neural Network.

Data Preprocessing

The original dataset was in JSON format and contained extensive, extraneous information related to sellers, which was deemed unnecessary for the analysis. Consequently, we filtered out the non-essential seller data, retaining only relevant information that provides meaningful insights about the vehicles. The processed data was then parsed and saved into separate CSV files. The implementation of this process can be found in the following section: `Limpiador.py`.

After filtering the data into a `.csv` file, we extract information from the detailed descriptions using regular expressions (regex) and standardize them. For instance, certain columns contain values such as `"25cm diametro"`, which we parse using regex and convert into standardized integer values, e.g., `25 (int)`.

Listing 1: Python Function for Processing Technical Details

```python
def process_technical_details(car: dict) -> dict:

    if "Ficha Tecnica" in car:
        # Concatenate the "Ficha Tecnica" list into a single string
        text = "#".join(car["Ficha Tecnica"])

        # Extract technical details using regular expressions
        car["jato_classification"] = extract_word(r'clasificacion JATO: .*?(\w+\d+)', text)
        car["traction"] = extract_word(r'Traccion\s+(\w+)', text)
        car["brakes"] = extract_phrase(r'[^#]*frenos[^#]*', text, 0)
        car["front_suspension"] = extract_phrase(r'Suspension delantera([^,]*)', text)
        car["rear_suspension"] = extract_phrase(r'suspension trasera([^#]*)', text)
        car["power_cv"] = extract_word(r'(\d+)\s*CV', text)
        car["power_kw"] = extract_word(r'(\d+)\s*kW', text)
        car["rpm_max_power"] = extract_word(r'([\d.,]+)\s*rpm\s*\(potencia max\)', text)
        car["max_torque_nm"] = extract_word(r'(\d+)\s*Nm', text)
        car["rpm_max_torque"] = extract_word(r'([\d.,]+)\s*rpm\s*\(par max\)', text)
        car["motor_description"] = extract_phrase(r'Motor de[^#]*', text, 0)
        car["transmission_description"] = extract_phrase(r'Transmision de tipo[^#]*', text, 0)

        # Remove the original "Ficha Tecnica" field
        car.pop("Ficha Tecnica")
    else:
        # Assign None if "Ficha Tecnica" is not present
        keys = [
            "jato_classification", "traction", "brakes", "front_suspension",
            "rear_suspension", "power_cv", "power_kw", "rpm_max_power",
            "max_torque_nm", "rpm_max_torque", "motor_description",
            "transmission_description"
        ]
        for key in keys:
            car[key] = None

    return car
```

In some cases, there were categorical columns represented as integers. We replaced these integer values with their corresponding string labels for better interpretability. For instance, the `bodyTypeId` column was mapped as follows:

Listing 2: Python Function for Assigning Body Type Labels

```python
def assign_body_type(car: dict) -> dict:
    body_type_id = car.get("bodyTypeId", None)
    match body_type_id:
        case None: car["bodyTypeId"] = None
        case 1: car["bodyTypeId"] = "Berlina"
        case 2: car["bodyTypeId"] = "Coupe"
        case 3: car["bodyTypeId"] = "Cabrio"
        case 4: car["bodyTypeId"] = "Familiar"
        case 5: car["bodyTypeId"] = "Monovolumen"
        case 6: car["bodyTypeId"] = "SUV"
        case 7: car["bodyTypeId"] = "Pick Up"
        case 8: car["bodyTypeId"] = "Furgoneta"
    return car
```

Several additional data adaptation implementations have been made, which are detailed in the source code. For further information, please refer to the following script: `Preproccessing1.py`.

After the previous processing step, there still are certain columns that cannot be handled directly using regular expressions. These columns consist of descriptions generated by combining several categorical values. Specifically, these columns can be classified into four main categories: `Acabado interior`, `Acabado exterior`, `Electrical features`, and `Confort`.

For these columns, we initially attempted to extract all numerical values. However, in cases where the numerical content was extensive or complex, we chose to retain the column in its original form, appending a `_description` suffix to the column name. This decision allows us to later apply Natural Language Processing (NLP) techniques for deeper analysis and extraction of relevant information. For columns that were amenable to direct processing, we extracted the most significant information and created new binary, numerical, or categorical columns based on the extracted data.

The resulting dataset, while significantly improved, is still not entirely complete, as it contains missing values (e.g., `NA` entries). We chose to retain these missing values at this stage because, in future algorithm implementations, different methods of handling missing data may be required depending on the specific needs of each model. For instance, during a regression task, the data could be split based on fuel type, and missing values could be imputed using regression techniques tailored to the characteristics of different columns. The full implementation of this process can be found in the following section: `Preprocessing2.py`.

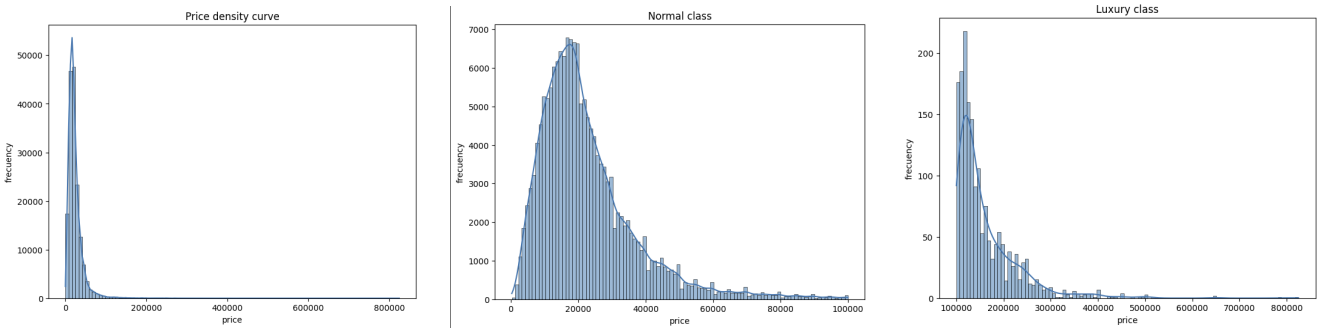
Listing 3: Fragments of the source code

```
data["Direccion_Asistida"] = data["Confort_Direccion"].apply(lambda x: True if 'direccion asistida'
    in x.lower() else False)
data["Direccion_Electrica"] = data["Confort_Direccion"].apply(lambda x: True if 'electrica' in
    x.lower() else False)
data["Endurecimiento_Progresivo"] = data["Confort_Direccion"].apply(lambda x: True if
    'endurecimiento progresivo' in x.lower() else False)
data["Desmultiplicacion_Variable"] = data["Confort_Direccion"].apply(lambda x: True if
    'desmultiplicacion variable' in x.lower() else False)
data["Direccion_Electro_Hidraulica"] = data["Confort_Direccion"].apply(lambda x: True if
    'electro-hidraulica' in x.lower() else False)

word_to_number = {
    "uno": "1", "dos": "2", "tres": "3", "cuatro": "4", "cinco": "5",
    "seis": "6", "siete": "7", "ocho": "8", "nueve": "9", "diez": "10",
    "once": "11", "doce": "12", "trece": "13", "catorce": "14", "quince": "15",
    "dieciseis": "16", "diecisiete": "17", "dieciocho": "18", "diecinueve": "19", "veinte": "20",
    "viente": "20", "vientecinco": "25", "vientecuatro": "24", "vienteuno": "21", "vientedos": "22",
    "vientetres": "23", "vienteseis": "26", "vientesiete": "27", "vienteocho": "28", "vientenueve":
    "29",
    "treinta": "30",

}

# 1. Function to extract engine displacement in liters (e.g., "1.6 liters")
def extract_displacement_liters(phrase):
    match = re.search(r"(\d+,\d+) litros", phrase)
    return match.group(1) if match else "NA"

# 2. Function to extract engine displacement in cubic centimeters (e.g., "1.582 cc")
def extract_displacement_cc(phrase):
    match = re.search(r"\(\s*([\d,.]+)\s*cc\s*\)", phrase)
    return match.group(1) if match else "NA"

# 3. Function to extract the number of cylinders (e.g., "four cylinders" or "4 cylinders")
def extract_number_of_cylinders(phrase):
    match = re.search(r"(\w+|\d+) cilindros", phrase)
    if match:
        cylinders = match.group(1).lower()
        return word_to_number.get(cylinders, cylinders)  # Convert word to number if necessary
    return "NA"

data["displacement_liters"] = data["motor_description"].apply(extract_displacement_liters)
data["displacement_cc"] = data["motor_description"].apply(extract_displacement_cc)
data["number_of_cylinders"] = data["motor_description"].apply(extract_number_of_cylinders)
```

Data Visualization

`DataVisualization`.

Now that the data is nearly clean, we are ready to proceed with data visualization. Let examine the distribution of our regression target variable, the price:

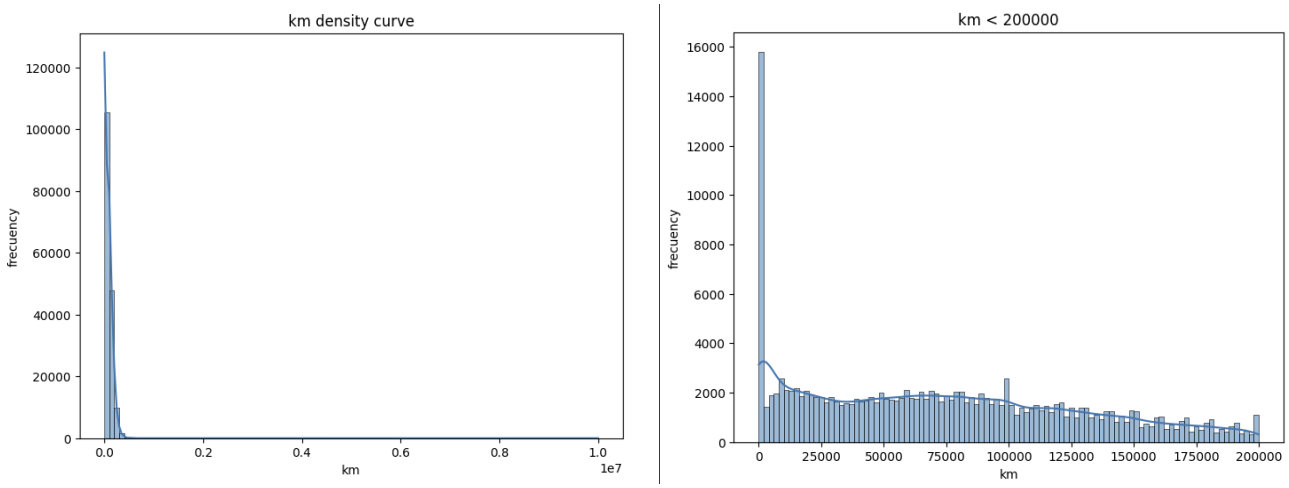


(a) Price Density Curve (b) Normal Price Density (c) Luxury Price Density

Figure 1: Price Density Curves

In Figure 1.(a), we observe that the distribution is highly skewed. For convenience, we shall divide the distribution at the \$100,000 mark. Vehicles priced under this threshold are categorized as `normal price`, the distribution of which is depicted in Figure 1.(b). Conversely, vehicles exceeding this price point are classified as `luxury`. The distribution for this category can be examined in Figure 1.(c).

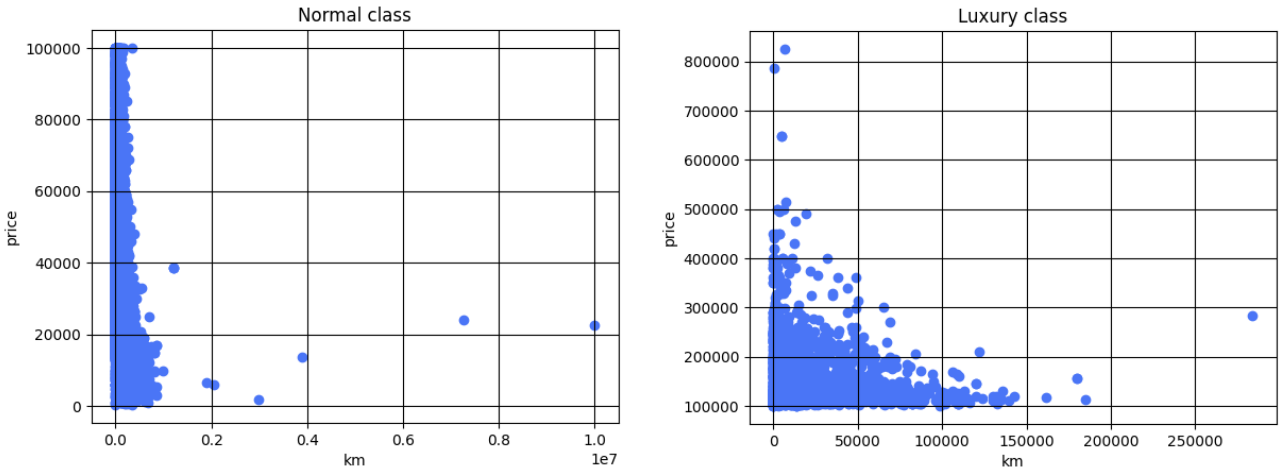
As the dataset is composed by second handed cars, lets visualize also the distribution of the km of the cars.



(a) Price Density Curve (b) Normal Price Density

Figure 2: Km Density Curves

We can confirm that most of the cars have odometer readings under 200,000 km, with a significant portion being 0 km vehicles. The distribution is even more skewed than that of the prices. Let's investigate whether there exists any linear relationship between the odometer reading (km) and the price.



(a) Normal price dot plot

(b) Luxury price dot plot

Figure 3: price dot plot

According to Figure 3, a clear relationship between the two variables, kilometers and price, cannot be established. This may be attributed to insufficient dimensionality. Consequently, we shall explore the potential impact of brand value on the pricing of second-hand cars. To facilitate this analysis, we will compare the modal price range (i.e., the most frequently occurring price range).

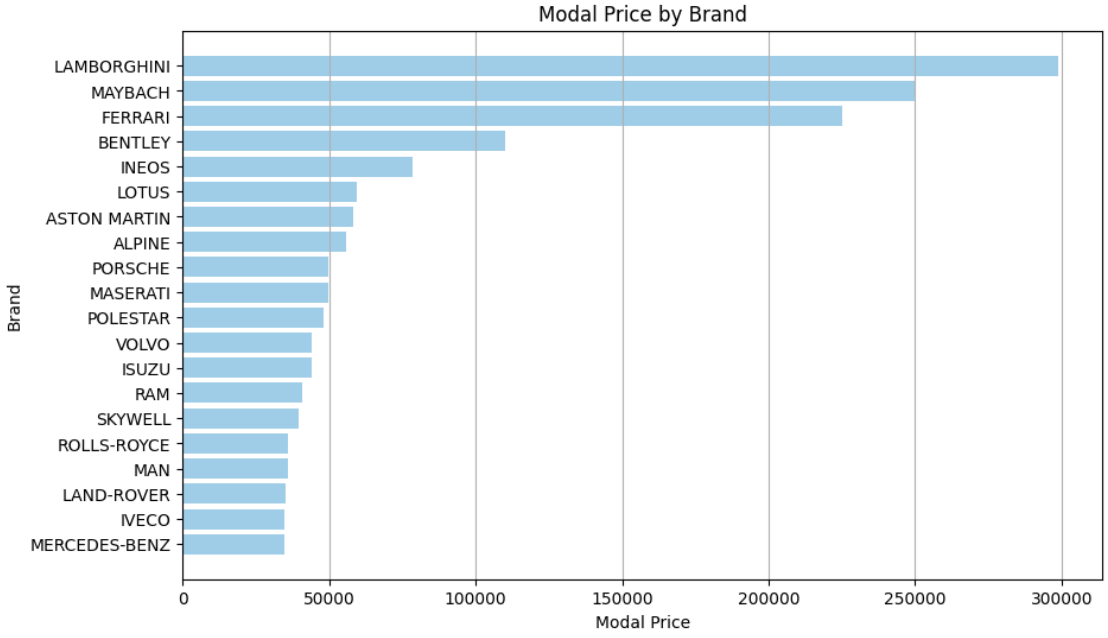
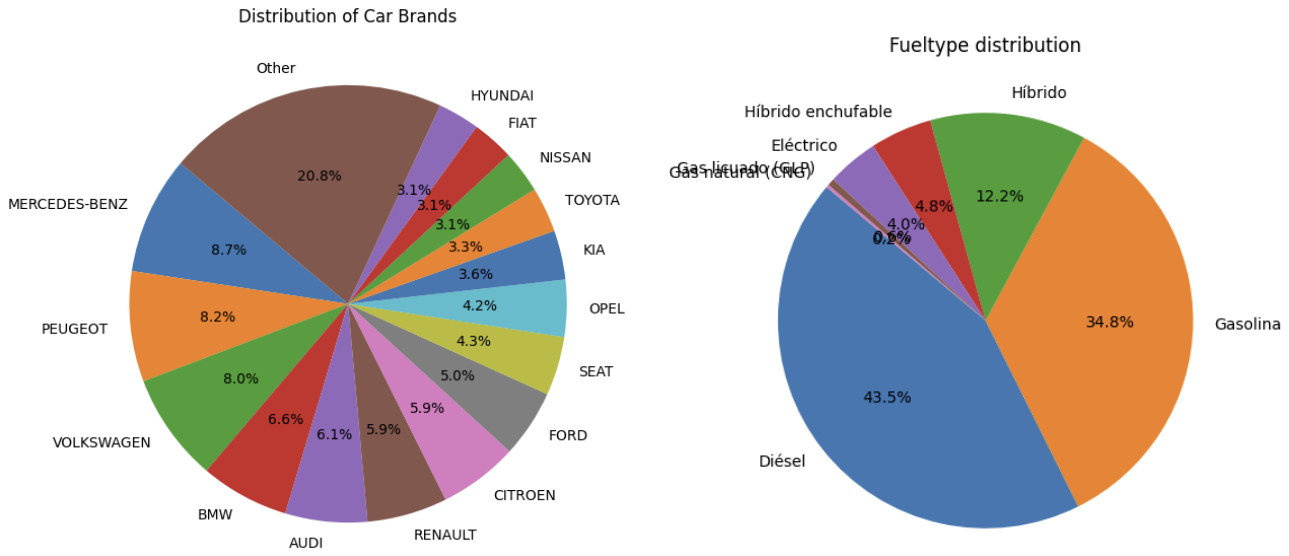


Figure 4: Modal price per brand Plot

Based on Figure 4, we can clearly affirm that some brands preserve their value in the second-hand market, now let's visualize which brands are more popular. Also, we must visualize which is the most popular fueltype in the second market.



(a) Brands pie chart

(b) Fueltypes pie chart

Figure 5: Fuels and brands distribution

Now that we have an overview of the data, we can proceed with processing it and extracting valuable information using the algorithms covered in our classes. Before diving into the analysis, we have added a new column called `price_categ`, which categorizes the cars based on their price using the following criteria:

Listing 4: Function to assign price category

```
def assign_price_categ(price: int) -> str:
    if   price < 5000:  return "Very low end"
    elif price < 10000: return "Low end"
    elif price < 15000: return "Budget"
    elif price < 25000: return "Middle low range"
    elif price < 30000: return "Middle range"
    elif price < 35000: return "Middle high range"
    elif price < 40000: return "High end"
    elif price < 50000: return "Premium"
    else: return "Luxury"
```

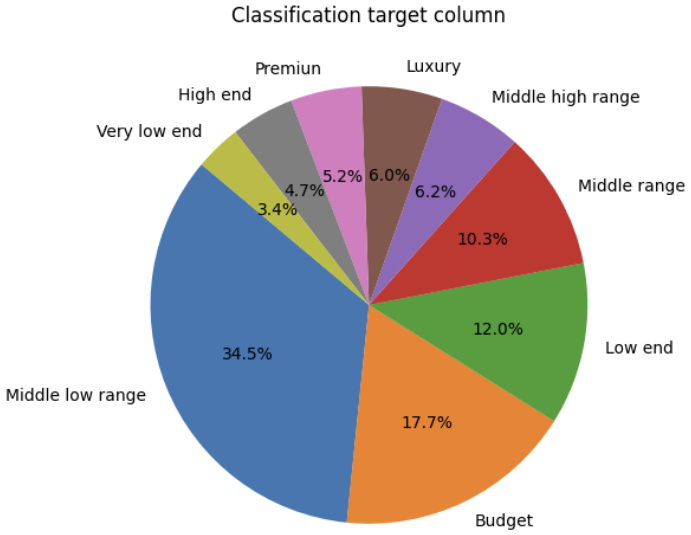


Figure 6: Modal price per brand Plot

Data processing

Let us use the electric vehicle dataset as an example to explain how we processed the data. The initial task involved checking for outliers. Due to the highly skewed distribution in the target column, a substantial number of outliers were detected. However, there were only a few examples at the extreme end of the distribution. Upon manual inspection, we confirmed that these data points were not outliers but rather represent very "eccentric" cars. (Figure.7)

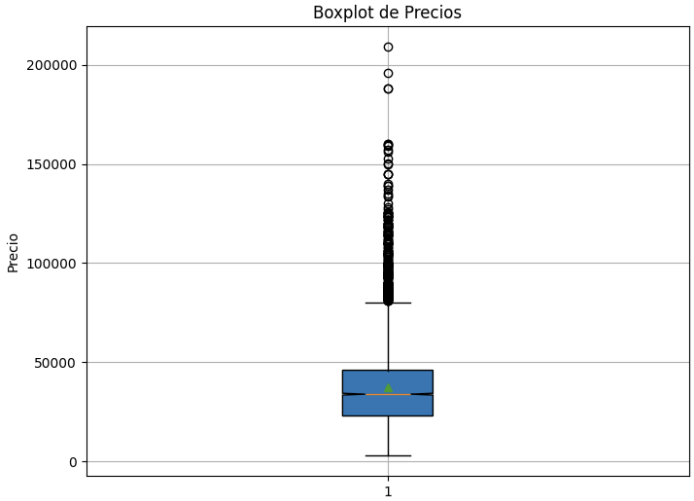


Figure 7: Price Box plot

After verifying the data, we need to clean all the missing values. To achieve this, we first split the columns into numerical and categorical columns. For the numerical columns, our strategy was to use regression to impute the missing data. To perform this imputation, we select the most correlated columns, denoted as X, to predict the target column Y, creating a function $f : X \to Y$. To do that, we used the correlation matrix, giving results such as:

- Regression MAPE [dimensions.length, dimensions.height] → dimensions.width: **1.6468%**
- Regression MAPE [power_cv, power_kw] → maxSpeed: **5.6895%**
- Regression MAPE [power_cv, power_kw, maxSpeed] → acceleration: **13.0356%**
- Regression MAPE [power_cv, power_kw, maxSpeed] → acceleration: **11.8549%**
- Regression MAPE [Llantas_Diametro_cm, power_kw] → max_torque_nm: **17.8541%**

After completing the numerical data imputation, we applied Principal Component Analysis (PCA) to reduce the dimensionality of the dataset (biplot result shown in Figure 8). Additionally, we used the correlation matrix to analyze the relationship between the features and our target variable. Based on these analyses, we decided to discard several variables that showed low contribution or high collinearity, including `Numero_Testigos`, `dimensions.height`, `doors`, `electricFeatures.onboardCharger_kW`, and `seatingCapacity`. (Figure.8)

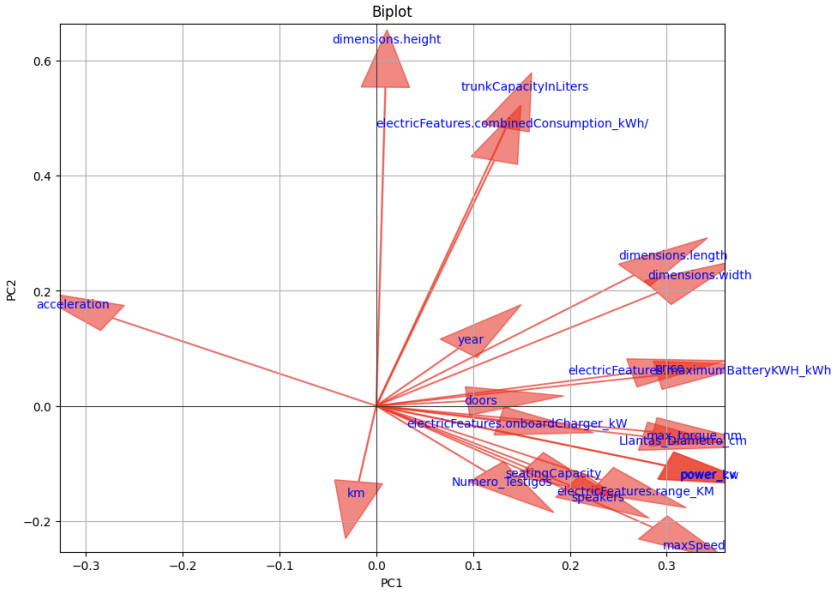


Figure 8: PCA Biplot

In order to filling missing values in categorical columns, we used a method called **impute categorical mode**. This method involves selecting a set of predictor columns X to impute missing values in the target column Y. To identify the most relevant columns X for imputing Y, we applied two main techniques:

- **Correspondence Analysis:** A method that reduces dimensionality for categorical data, enabling us to identify relationships between categorical variables.
- **Chi-Square Test of Independence:** This test was applied to each pair of categorical variables to quantify their association and identify the columns X that are most strongly associated with Y.

After selecting the relevant columns in X for imputation, we ordered them from the least correlated to the most correlated with Y. This ordering helps in grouping and imputing the missing values hierarchically. The imputation method follows a sequential grouping approach based on the columns in X:

1. **Iterative Grouping and Imputation:** We start by grouping the rows based on the least correlated columns in X, progressing sequentially to the most correlated ones. For each group, we impute missing values in Y using the most frequent value (mode) within that group.
2. **Dimensionality Reduction:** In each iteration, we remove the least important column in X, reducing the dimensionality of the grouping. This approach allows us to impute missing values in Y by first using more granular groupings, followed by broader groupings as we progress.

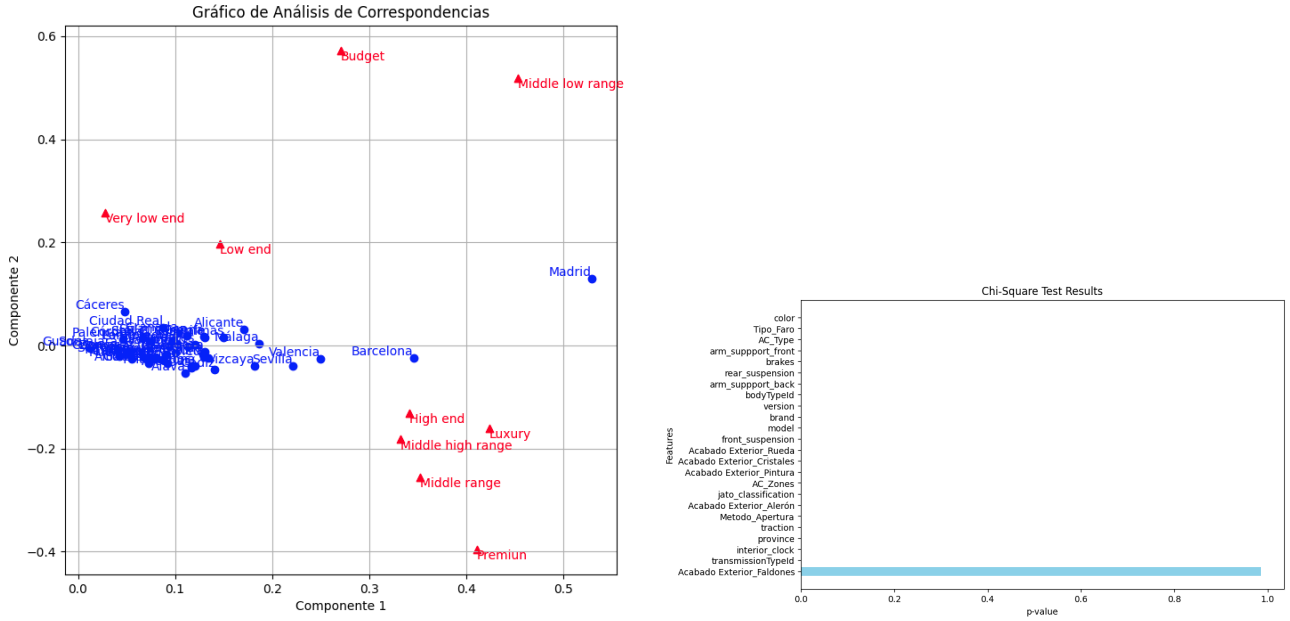
If there are still missing values in Y after all groupings, we assign the default value `"unknown"`.

The following code implements the imputation process:

Listing 5: Categorical value imputation

```
def __impute_categorical_mode(df, X, Y):
    modes = df.groupby(X, observed=False)[Y].agg(
        lambda x: x.dropna().mode()[0] if not x.dropna().empty else None
    )
    modes.name = 'Mode'
    df = df.join(modes, on=X, how='left')
    df[Y] = df.apply(lambda row: row['Mode'] if pd.isna(row[Y]) else row[Y], axis=1)
    df.drop('Mode', axis=1, inplace=True)
    return df
def impute_categorical_mode(df, X, Y):
    for i in range(len(X)):
        df = Data_processor.__impute_categorical_mode(df, X, Y)
        X.pop(len(X) - 1)
    df[Y] = df[Y].fillna("unknown")
    return df
```

Once the missing values is filtered we perform again chi-square test and drop the columns whose p-value is almost 0. Leaving us the result that can be observed in the Figure.9



(a) Price Density Curve

(b) Normal Price Density

Figure 9: Price Density Curves

After processing the data we standarize the numerical values and create dummy columns of categorical columns, then we applied the algorithms that we have seen in class. Leaving us with this result:

Algorithm	MAE	MSE	MAPE	R^2	Error Mean	Error Std Dev	Adjusted R^2
CART	3.5598e+03	4.4347e+07	9.9821e+00	8.9670e-01	3.5598e+03	5.6280e+03	1.0283e+00
RandomForest	2.8870e+03	2.3943e+07	8.3304e+00	9.4420e-01	2.8870e+03	3.9507e+03	1.0153e+00
SVR	1.4729e+04	4.3164e+08	5.0342e+01	-5.2000e-03	1.4729e+04	1.4652e+04	1.2758e+00
Lineal Regression	7.6029e+12	2.3274e+27	2.7033e+10	-5.4199e+18	7.6029e+12	4.7640e+13	1.4872e+18
ANN	3.3902e+03	2.8687e+07	9.7746e+00	9.3320e-01	3.3902e+03	4.1464e+03	1.0183e+00

Table 2: Comparison of Regression Models (Best results highlighted)

Algorithm	accuracy	precision	recall	f1
CART	6.9570e-01	6.9720e-01	6.9570e-01	6.9600e-01
RandomForestClassifier	7.1280e-01	7.1420e-01	7.1280e-01	7.1250e-01
SVC	6.5370e-01	6.6670e-01	6.5370e-01	6.5330e-01
Naive bayes	4.8740e-01	5.7810e-01	4.8740e-01	4.7020e-01
ANN	6.8550e-01	6.8610e-01	6.8550e-01	6.8510e-01

Table 3: Comparison of Classification Models (Best results highlighted)

At the first sight, RandomForest is surprisingly good on both task, so in order to verify that it is not overfitted we perform a cross-folder validation. Lineal regression and SVM are significantly the worst in both task, due to the huge number of dimensions in the data, before creating dummy columns we had 28 and after creating dummy columns we had 9128 (Figure.10)

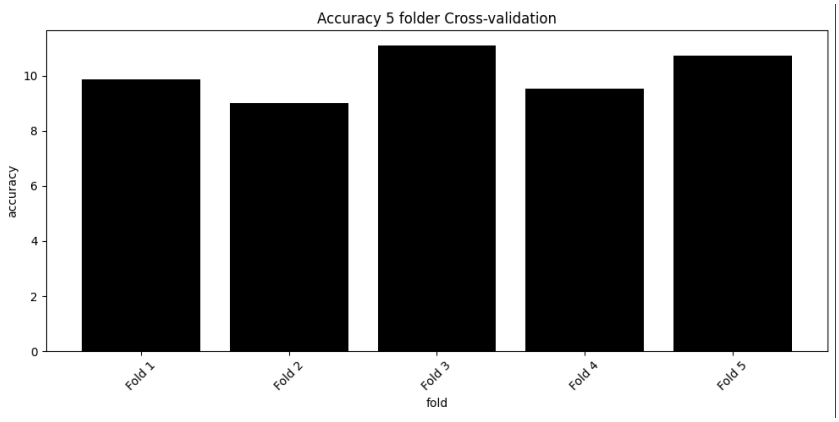
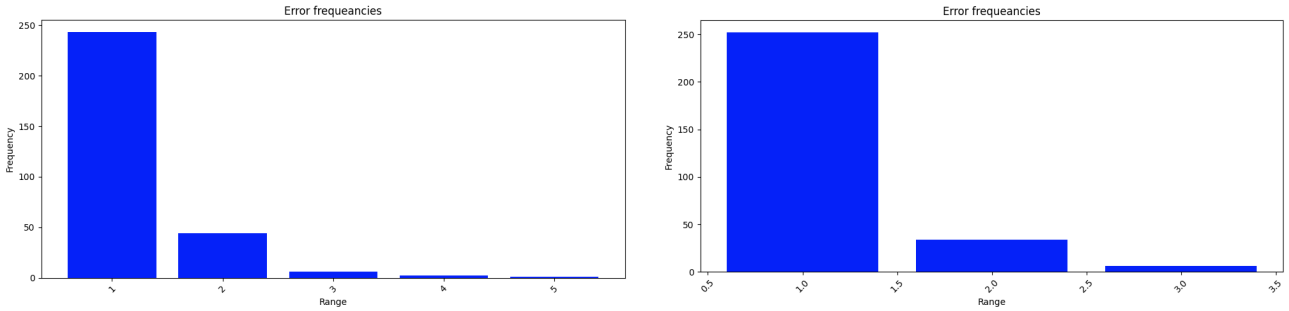


Figure 10: Cross Validation Accuracy

The target column, `price_tag`, in the classification task contains ordinal categorical values. This allows us to evaluate the significance of the errors made by the models. Although both CART and Random Forest show similar values in terms of accuracy, precision, recall, and F1 score, we can further assess and compare their capacity to differentiate between the ordinal categories (see Figure 11).



(a) CART

(b) Random Forest

Figure 11: CART vs Random Forest

Both algorithms exhibit a similar number of errors; however, one demonstrates a significantly higher capacity for differentiation, as its errors are smaller. Specifically, an error of 1 degree may indicate that the model confused a "very low-end" category with a "low-end" category. In contrast, an error of 6 degrees suggests that the model cannot distinguish between a "very low-end" category and a "high-end" category.

Source code: `electric_and_plugin`, `combustion`, `hybrid` and `gas`.

As stated before, we repeated a similar process to all the cars in each category of `fuel_type`. The results were the following:

Algorithm	accuracy	precision	recall	f1
CART	7.3320e-01	7.3360e-01	7.3320e-01	7.3340e-01
RandomForestClassifier	7.9250e-01	7.8910e-01	7.9250e-01	7.9010e-01
Naive bayes	5.3450e-01	5.3830e-01	5.3450e-01	5.1540e-01
ANN	7.7260e-01	7.7130e-01	7.7260e-01	7.7180e-01

Table 4: Classification Models for Cars with the **Combustion** Fuel Type (Best results highlighted)

Algorithm	mae	mse	mape	R^2	error mean	error std dev	adjusted R^2
CART	2.5741e+03	6.0937e+07	1.3611e+01	8.6250e-01	2.5741e+03	7.3696e+03	8.6230e-01
RandomForest	1.9632e+03	2.9673e+07	1.0814e+01	9.3300e-01	1.9632e+03	5.0812e+03	9.3300e-01
SVR	7.7928e+03	4.0261e+08	4.0963e+01	9.1500e-02	7.7928e+03	1.8490e+04	9.0400e-02
ANN	2.2742e+03	3.1866e+07	1.2473e+01	9.2810e-01	2.2742e+03	5.1666e+03	9.2800e-01
Lineal Regression	2.2742e+03	3.1866e+07	1.2473e+01	9.2810e-01	2.2742e+03	5.1666e+03	9.2800e-01

Table 5: Regression Models for Cars with the **Combustion** Fuel Type (Best results highlighted)

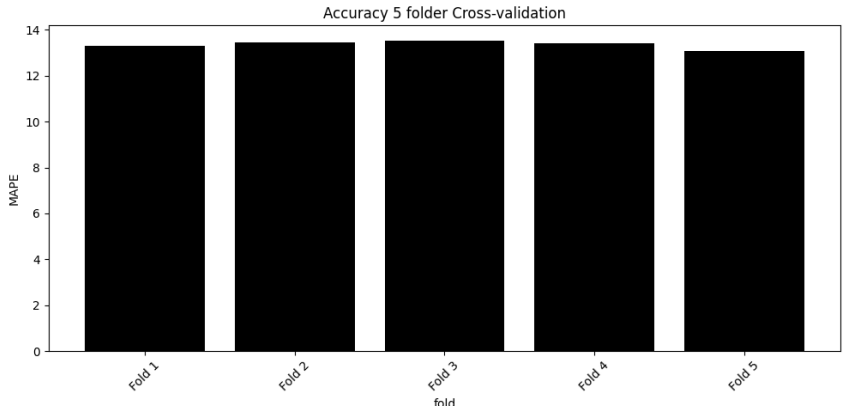
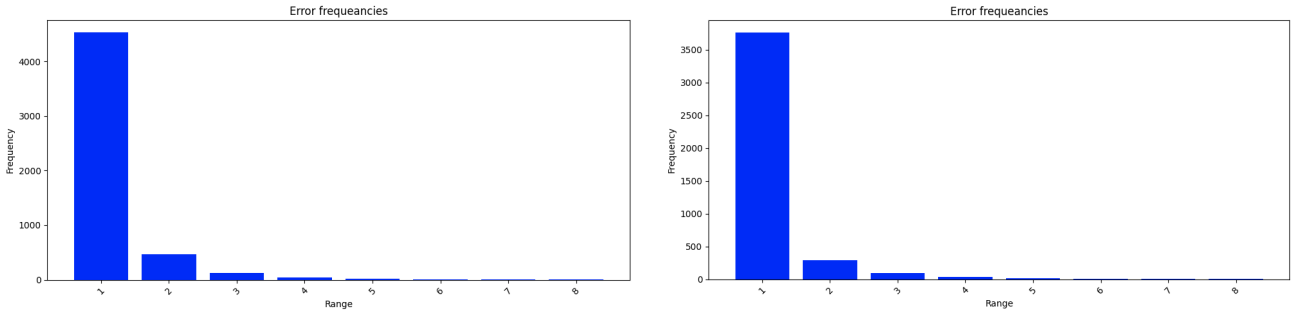


Figure 12: Cross Validation Accuracy **Combustion**



(a) CART (b) Random Forest

Figure 13: CART vs Random Forest **Combustion**

Algorithm	accuracy	precision	recall	f1
CART	8.0600e-01	8.0590e-01	8.0600e-01	8.0580e-01
RandomForestClassifier	8.2370e-01	8.2370e-01	8.2370e-01	8.2330e-01
SVC	7.8670e-01	7.9110e-01	7.8670e-01	7.8450e-01
Naive bayes	4.6450e-01	6.0620e-01	4.6450e-01	4.6360e-01
ANN	7.9020e-01	7.9280e-01	7.9020e-01	7.9110e-01

Table 6: Classification Models for Cars with the **Hybrid** Fuel Type (Best results highlighted)

Algorithm	mae	mse	mape	R^2	error mean	error std dev	adjuste R^2
CART	1.9425e+03	1.5163e+07	5.9639e+00	9.4970e-01	1.9425e+03	3.3749e+03	8.1700e-01
RandomForest	1.5919e+03	8.5062e+06	4.9645e+00	9.7180e-01	1.5919e+03	2.4438e+03	8.9730e-01
SVR	1.0531e+04	3.0604e+08	3.3087e+01	-1.5700e-02	1.0531e+04	1.3969e+04	-2.6932e+00
Lineal Regression	3.8011e+12	3.3859e+26	1.4128e+10	-1.1237e+18	3.8011e+12	1.8004e+13	-4.0860e+18
ANN	1.7955e+03	9.3936e+06	5.6205e+00	9.6880e-01	1.7955e+03	2.4839e+03	8.8660e-01

Table 7: Regression Models for Cars with the **Hybrid** Fuel Type (Best results highlighted)

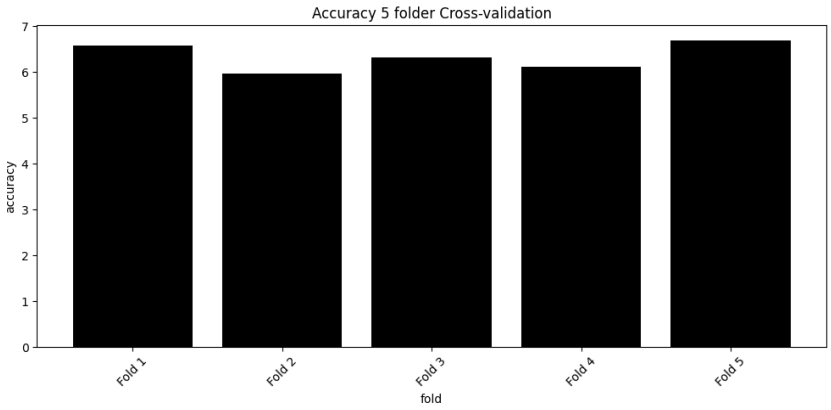
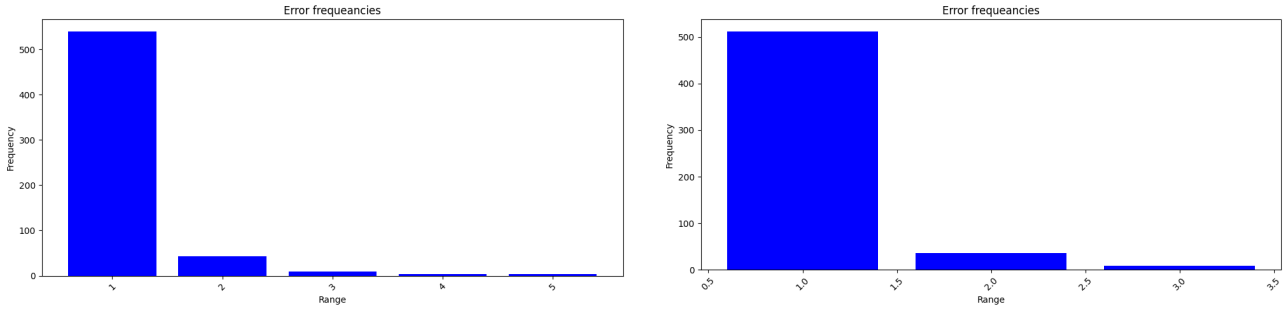


Figure 14: Cross Validation Accuracy **Hybrid**



(a) CART (b) Random Forest

Figure 15: CART vs Random Forest **Hybrid**

Algorithm	accuracy	precision	recall	f1
CART	8.4110e-01	8.6080e-01	8.4110e-01	8.4950e-01
RandomForestClassifier	8.8760e-01	8.9300e-01	8.8760e-01	8.8850e-01
SVC with SMOTE	4.2250e-01	5.9090e-01	4.2250e-01	4.4600e-01
SVC	6.7050e-01	6.8710e-01	6.7050e-01	6.6290e-01
Naive bayes	7.1710e-01	7.5470e-01	7.1710e-01	7.2700e-01
ANN	8.8630e-01	8.9600e-01	8.8630e-01	8.8750e-01

Table 8: Classification Models for Cars with the **Gas** Fuel Type (Best results highlighted)

Algorithm	mae	mse	mape	r2	error mean	error std dev	adjusted R^2
CART	7.7868e+02	1.7568e+06	5.4355e+00	9.3920e-01	7.7868e+02	1.0726e+03	1.0028e+00
RandomForest	6.9823e+02	1.1917e+06	4.7292e+00	9.5880e-01	6.9823e+02	8.3913e+02	1.0019e+00
SVR	4.2166e+03	2.9860e+07	2.9525e+01	-3.2600e-02	4.2166e+03	3.4756e+03	1.0468e+00
ANN	9.2740e+02	1.6381e+06	6.4002e+00	9.4340e-01	9.2740e+02	8.8206e+02	1.0026e+00

Table 9: Regression Models for Cars with the **Gas** Fuel Type (Best results highlighted)

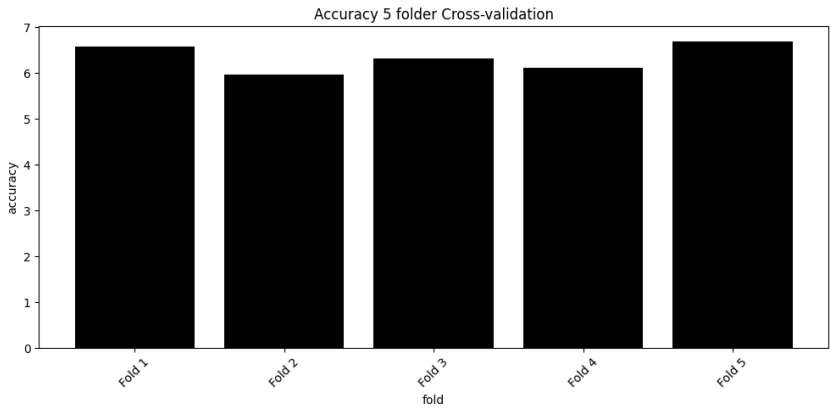
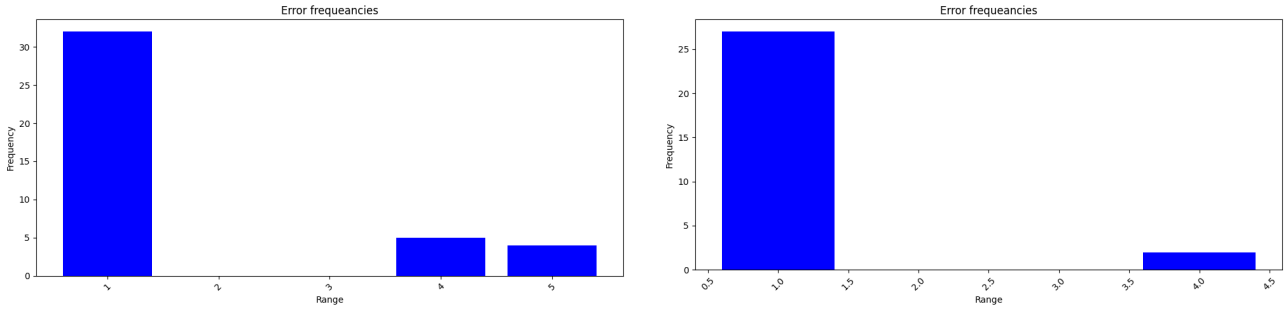


Figure 16: Cross Validation Accuracy **Hybrid**

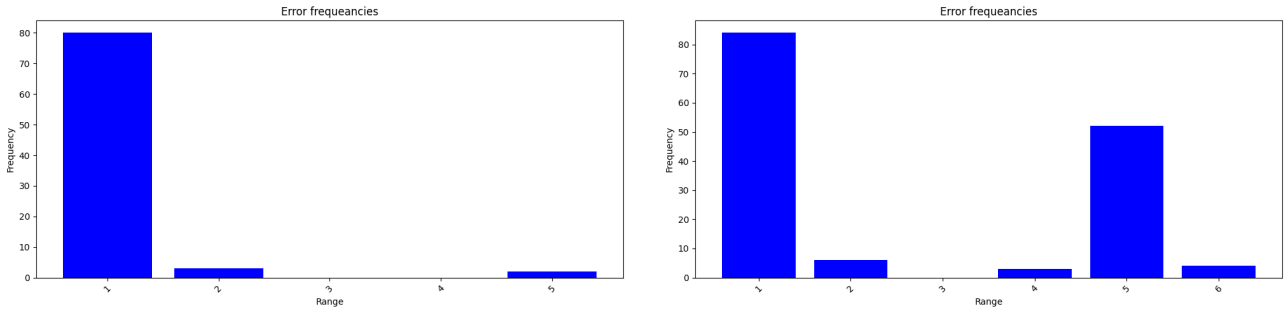


(a) CART (b) Random Forest

Figure 17: CART vs Random Forest **Gas**

Class Imbalance Analysis and SMOTE Application

For the data of type "gas", it was observed that the classes were somewhat imbalanced, which initially led us to apply SMOTE to address this issue. SMOTE was chosen because it effectively balances datasets by generating synthetic samples for the minority class. However, after analyzing the results obtained with and without the application of SMOTE, it was found that models trained without SMOTE performed better in terms of accuracy and stability. As a result, the remainder of the models were developed without using this technique.



(a) SVM (b) SVM with SMOTE

Figure 18: SVM vs SVM with SMOTE **Gas**

Results without hyperparameter tunning

Based on the analysis results, the Random Forest model demonstrates superior performance in both the regression and classification tasks. This can be attributed to the non-linear nature of the data, as observed during data visualization, and the high dimensionality of the dataset. Additionally, the model's ability to assess feature importance contributes to its effectiveness, providing valuable insights into the relevance of different variables.

To analyze the data in more detail, three different techniques were used: regression, classification, and clustering. These methods were selected to examine various aspects of the dataset and to confirm the reliability of the previous findings.

Regression after hyperparameter tuning

After the extensive hyperparameter tuning process using Optuna, we have obtained the results shown in Tables 10-12. Overall, there is a very noticeable improvement in CART and RF, and they fit significantly better on medium-sized datasets. On larger datasets (such as the combustion cars dataset), the models demonstrate better performance. (The best results are highlighted in yellow.)

The most significant improvement is observed in the SVR, which performs quite poorly without hyperparameter tuning but delivers much more notable results after tuning. However, in terms of computational cost, it is far less efficient compared to trees or a Fully Connected neural network. In fact, its cost-benefit ratio is even worse than that of using NLP techniques.

Decision Tree hyperparameters in regression task

CART	Hybrid	Electric and Plugin	Gas	Combustion
max_depth	27	22	11	30
textbfmin_sample_split	2	2	11	2
msl	1	1	7	1
Mf	NaN	NaN	NaN	NaN
criterion	fr_mse	abs_error	abs_error	fr_mse
mae	1.8362e+03	3.4531e+03	1.0825e+03	3.1577e+03
mse	1.2834e+07	4.3935e+07	2.6471e+06	7.5457e+07
mape	5.6104e+00	9.1234e+00	7.6819e+00	1.6970e+01
r2	9.5610e-01	9.0150e-01	9.0150e-01	8.1030e-01
error_mean	1.8362e+03	3.4531e+03	1.0825e+03	3.1577e+03
error_std_dev	3.0761e+03	5.6578e+03	1.2146e+03	8.0923e+03
adjusted_r2	1.5008e+00	1.0155e+00	1.0029e+00	8.0980e-01

Table 10: mss = min samples split, msl = min samples leaf, Mf = max features, Md = max depth

RandomForest in regression task

RF	Hybrid	Electric and Plugin	Gas	Combustion
n_estimator	100	200	100	400
max_depth	478	494	445	228
min_samples_split	20	3	3	2
min_samples_leaf	5	1	1	1
max_features	log2	NaN	NaN	log2
bootstrap	TRUE	TRUE	TRUE	TRUE
mae	8.0301e+03	2.9701e+03	8.3023e+02	2.5204e+03
mse	1.4863e+08	2.4026e+07	1.6911e+06	3.9615e+07
mape	2.9206e+01	8.1902e+00	6.0631e+00	1.4894e+01
r2	4.9200e-01	9.4940e-01	9.3710e-01	9.0040e-01
error_mean	8.0301e+03	2.9701e+03	8.3023e+02	2.5204e+03
error_std_dev	9.1731e+03	3.8993e+03	1.0009e+03	5.7674e+03
adjuste_r2	6.7997e+00	1.0085e+00	1.0019e+00	9.0010e-01

Table 11:

SVM in regression task

SVR	Hybrid	Electric and Plugin	Gas	Combustion
kernel	rbf	poly	poly	poly
C	83.784313	82.266313	84.685	79.790
epsilon	0.387	0.284219	0.499	0.499
degree	5	5	5.000	5.000
coef0	0.999	0.999	0.997	0.999
gamma	scale	scale	scale	scale
max_iter	2000	3000	5000	6000
mae	5.4433e+03	3.4091e+03	9.3625e+02	2.3911e+03
mae	5.4433e+03	3.4091e+03	9.3625e+02	2.3911e+03
mse	1.2566e+08	2.7973e+07	1.8653e+06	3.5369e+07
mape	1.6487e+01	9.4796e+00	6.6341e+00	1.2181e+01
r2	5.7810e-01	9.4110e-01	9.3060e-01	9.1110e-01
error_mean	5.4433e+03	3.4091e+03	9.3625e+02	2.3911e+03
error_std_dev	9.7998e+03	4.0436e+03	9.9433e+02	5.4453e+03
adjuste_r2	-5.3410e-01	1.0099e+00	1.0021e+00	9.1080e-01

Table 12:

Regression models interpretation

In the folder available at Github, you can find images and PDF files illustrating the structure of the trees. It is worth noting that in both *RandomForest* and *CART* for regression tasks, the first node is always related to the car's power (`power_cv` in RF and `power_kW` in CART). At subsequent levels, attributes such as mileage, car dimensions, and brand appear, which can be understood as the order of priorities for a customer when purchasing a used car.

These factors are the most decisive when setting a car's price, following the logic of supply and demand. Thus, they represent the most important aspects for a potential buyer. In contrast, parameters related to a car's comfort features, such as seat heating, are located at the lower levels of the tree. Consequently, they are not significant when determining the car's price.

Classification models after hyperparameter tuning

In classification, the hyperparameter tuning follows the same approach as the regression models discussed earlier, yielding a very significant improvement, especially in the *SVC*, which benefits tremendously. On smaller datasets, slight overfitting is observed, as the gas car dataset is exceptionally small. Then, on medium-sized datasets, there is a notable improvement in the models' accuracy, and the same applies to larger datasets.

However, the *SVC* fails to adapt adequately to the largest datasets. Despite the notable improvement, it still performs worse than the worst of the estimations, that is, the mean.

Decision Tree hyperparameters in classification task

CART	Hybrid	Electric and Plugin	Gas	Combustion
Md	28	23	6	20
mss	6	3	15	2
msl	1	1	8	1
accuracy	0.963682	0.951478	0.829457	0.883712
precision	0.963867	0.952858	0.839629	0.882964
recall	0.963682	0.951478	0.829457	0.883712
f1	0.963680	0.951547	0.831693	0.883060

Table 13: mss = min samples split, msl = min samples leaf, Mf = max features, Md = max depth

Random Forest hyperparameters in classification task

RF	Hybrid	Electric and Plugin	Gas	Combustion
n_estimator	350	350	100	528
Md	434	325	450	289
mss	3	3	2	6
msl	1	1	1	2
accuracy	0.992537	0.974981	1.000000	0.902248
precision	0.992543	0.975236	1.000000	0.901392
recall	0.992537	0.974981	1.000000	0.902248
f1	0.992533	0.975015	1.000000	0.901163

Table 14: mss = min samples split, msl = min samples leaf, Mf = max features, Md = max depth

SVC hyperparameters in classification task

SVC	Hybrid	Electric and Plugin	Gas	Combustion
C	0.680637	14.323218	24.851842	12.235245
kernel	linear	rbf	poly	rbf
gamma	auto	scale	scale	scale
accuracy	0.886816	0.877938	0.976744	0.502264
precision	0.887101	0.878135	0.976601	0.505432
recall	0.886816	0.877938	0.976744	0.502264
f1	0.886102	0.877938	0.976644	0.495419

Table 15: mss = min samples split, msl = min samples leaf, Mf = max features, Md = max depth

Source code: `electric_and_plugin`, `combustion`, `hybrid` and `gas`.

Interpretation of the classification models

Among the three models for which we have tuned the hyperparameters, the trees are the most interpretable. In classification, they follow the same pattern as described in regression; however, the root node, instead of being the engine power, shifts to the suspension of the rear wheels. Engine power then takes a secondary position, alongside mileage, car dimensions, and brand. (Reference information can be found in the GitHub repository, as the tree images are too large to include in the documentation.)

From this, it can be concluded that trees are an excellent algorithm for sales analysis, helping identify the most important factors of a product in relation to its price. *SVMs* are also effective, but they are far less interpretable. Unlike trees, where each parameter is clearly classified at a specific level, *SVMs* lack such structure. The same issue arises when comparing them to deep learning models, which are essentially black boxes.

Explanation of classification results

Looking at these results, something stands out: Random Forest performed exceptionally well, especially with gas cars, achieving 100% accuracy. However, this result raises some concerns, as it may indicate overfitting, particularly because the dataset contains relatively few gas cars. Gas cars generally follow more standardized pricing patterns, with years of market data and well-defined factors such as mileage, brand, model, and year influencing their prices. This could explain why the model performs well, but the limited sample size suggests caution in interpreting this accuracy.

Hybrids also showed strong prediction accuracy across all models, particularly with Random Forest (99%) and CART (96%). [Tables 13-14] This suggests that hybrid car prices follow relatively predictable patterns, probably because they're now well-established in the market with clear value propositions based on fuel efficiency and features.

The most challenging category was combustion vehicles, especially for the SVC model (50% accuracy). [Table 15] This poor performance likely reflects the huge variety in traditional combustion cars – from cheap old vehicles to luxury sports cars – making price predictions more complex.

Hyperparameter analysis

The hyperparameter settings reveal interesting insights about the complexity needed for each vehicle category. For Random Forest, both hybrid and electric vehicles required substantial computational power with 350 trees and deep decision paths (max_depth 325-434). This suggests these newer vehicle categories have intricate pricing patterns that need detailed feature analysis.

CART's simpler structure used notably shallower trees, particularly for gas vehicles (max_depth 6), while hybrids needed deeper trees (max_depth 28). This aligns with market reality – gas vehicles often have more straightforward pricing based on well-established factors, while hybrid pricing involves more complex feature interactions.

The SVC model's kernel choices are particularly telling: gas vehicles performed best with a polynomial kernel, suggesting non-linear price relationships, while hybrids worked well with a linear kernel, indicating more standardized pricing structures. This makes sense given that hybrid vehicles often follow more consistent pricing patterns within their market segments.

The consistently low minimum samples split/leaf values (1-6) across models indicate very specific decision boundaries were needed, especially for newer vehicle technologies where small feature differences can significantly impact price category. This granularity requirement highlights the complexity of modern vehicle pricing, particularly in emerging categories like electric and hybrid vehicles.

Clustering

Finally, clustering techniques were used to discover hidden patterns and groupings in the dataset. These unsupervised learning methods reveal structures that are not obvious from other analyses.

The following sections provide a detailed discussion of the results for each technique and how they contribute to a better understanding of the dataset.

Clustering Results

Clustering Hyperparameters and Metrics

K-Means

Metric/Hyperparameter	Hybrid	Gas	Combustion	Electric and Plugin
n_clusters	4	4	2	4
init	kmeans++	kmeans++	kmeans++	kmeans++
n_init	**90**	23	42	47
max_iter	203	378	326	387
silhouette_score	0.1655	0.2461	0.1938	0.2205
calinski_harabasz_score	3901.5553	279.4322	5038.3059	1979.57
davies_bouldin_score	1.7228	1.4445	1.9938	1.4363
adjusted_rand_score	0.2333	0.1489	0.0651	0.1825
normalized_mutual_info_score	0.3105	0.1594	0.1009	0.2386
homogeneity_score	0.2433	0.1549	0.0684	0.1949
completeness_score	0.4288	0.1641	0.1925	0.3074
v_measure_score	0.3105	0.1594	0.1009	0.2386

Table 16: K-Means hyperparameters and metrics.

Agglomerative Clustering

Metric/Hyperparameter	Hybrid	Gas	Combustion	Electric and Plugin
n_clusters	4	4	2	4
linkage	single	single	complete	single
metric	manhattan	ward	ward	ward
silhouette_score	0.69	0.3353	0.952994	0.3688
calinski_harabasz_score	203.2587	80.6946	765.2020	16.8972
davies_bouldin_score	0.1791	0.6810	0.03283	0.5524
adjusted_rand_score	0.0000	0.0167	0.000038	0.0003
normalized_mutual_info_score	0.0003	0.0586	0.0014	0.0023
homogeneity_score	0.0002	0.0338	0.290966	0.0012
completeness_score	0.1459	0.2216	0.2909	0.1974
v_measure_score	0.0003	0.0586	0.000152	0.0023

Table 17: Agglomerative Clustering hyperparameters and metrics.

BIRCH

Metric/Hyperparameter	Hybrid	Gas	Combustion	Electric and Plugin
threshold	3	0.637	0.915036	0.917
n_clusters	0.9	4	2	3
branching_factor	76	82	15	78
silhouette_score	0.3847	0.2454	0.1330	0.2232
calinski_harabasz_score	1771.6270	266.2893	3448.3426	1581.0480
davies_bouldin_score	0.7302	1.4448	2.4287	1.4369
adjusted_rand_score	0.0284	0.1618	0.1164	0.0488
normalized_mutual_info_score	0.0756	0.1724	0.1407	0.1012
homogeneity_score	0.0418	0.1686	0.1047	0.0700
completeness_score	0.3954	0.1764	0.2147	0.1827
v_measure_score	0.0756	0.1724	0.1407	0.1012

Table 18: BIRCH hyperparameters and metrics.

DBSCAN

Metric/Hyperparameter	Hybrid	Gas	Combustion	Electric and Plugin
eps	0.5	NA	4.414662	0.5
min_samples	29	20	4	20
metric	euclidean	manhattan	euclidean	euclidean
algorithm	auto	auto	brute	auto
silhouette_score	-0.2548	0.0424	-0.2629	-0.3382
calinski_harabasz_score	45.9968	21.7083	12.3382	23.2894
davies_bouldin_score	1.1575	1.0114	1.4228	1.3358
adjusted_rand_score	-0.0005	0.0157	0.0005	0.0001
normalized_mutual_info_score	0.0214	0.0445	0.0044	0.0828
homogeneity_score	0.0112	0.0239	0.0022	0.0520
completeness_score	0.2344	0.3311	0.1905	0.2027
v_measure_score	0.0214	0.0445	0.0044	0.0828

Table 19: DBSCAN hyperparameters and metrics.

Gaussian Mixture Model (GMM)

Metric/Hyperparameter	Hybrid	Gas	Combustion	Electric and Plugin
n_components	7	40	2	7
covariance_type	full	diag	diag	full
tol	1e-3	0.0094	0.008958	1e-3
reg_covar	1e-6	0.0009	1e-6	1e-6
max_iter	200	190	109	200
silhouette_score	0.1397	0.2962	0.0662	0.1431
calinski_harabasz_score	2569.8803	121.0902	2145.2465	1415.2785
davies_bouldin_score	2.1590	1.3774	2.6886	1.8646
adjusted_rand_score	0.1696	0.0712	0.1041	0.1691
normalized_mutual_info_score	0.2969	0.2809	0.1860	0.2653
homogeneity_score	0.2957	0.5461	0.1882	0.2615
completeness_score	0.2981	0.1891	0.1839	0.2692
v_measure_score	0.2969	0.2809	0.1860	0.2653

Table 20: Gaussian Mixture Model hyperparameters and metrics.

OPTICS

Metric/Hyperparameter	Hybrid	Gas	Combustion	Electric and Plugin
min_samples	20	20	17	20
max_eps	10	3.721347	4.950154	5
metric	euclidean	euclidean	euclidean	euclidean
cluster_method	dbscan	dbscan	dbscan	dbscan
silhouette_score	0.8628	0.0813	0.6769	0.4952
calinski_harabasz_score	170.7215	71.8657	511.3782	28.2932
davies_bouldin_score	1.5547	1.8790	2.1182	1.1329
adjusted_rand_score	0.0000	0.0704	0.0007	0.0002
normalized_mutual_info_score	0.0002	0.1778	0.0021	0.0017
homogeneity_score	0.0001	0.2264	0.0011	0.0008
completeness_score	0.1028	0.1464	0.0689	0.1965
v_measure_score	0.0002	0.1778	0.0021	0.0017

Table 21: OPTICS hyperparameters and metrics.

Challenges in Obtaining Optimal Clusters

The Hopkins statistic for our dataset was calculated to be 0.9999999942512605, indicating a strong tendency for clustering. However, we believe that the clustering results did not meet expectations, primarily due to the following reasons:"

1. **Suboptimal Hyperparameter Ranges:** The clustering algorithm failed to explore the most effective hyperparameters, as they were not within the search space defined by Optuna. This limited the potential of the clustering method.

2. **Impact of Dataset Pre-Splitting:** The dataset was pre-split into four categories, which grouped the most distinguishable features. As a result, the remaining data lacked clear distinctions, making it challenging to identify well-defined clusters.

3. **Residual Data Complexity:** Despite the high Hopkins statistic, practical clustering was hindered by noise, overlapping feature distributions, and high dimensionality in the residual data.

4. **Lack of processing power:** Due to hardware limitations and the big amount of data, specially in the Combustion dataset, we had to limit it in two ways. First, we had to apply the algorithms on the components of an PCA that explains 95% of the data along with TruncatedSVD for categorical columns. Finally, due to time limitations Optuna was set with a low number of trials which may led to results that are not optimal.

Clustering Conclusion

According to Hopkins Statistics, there should be clearly separated clusters, which implies that effective clustering should yield a high silhouette score. However, in most methods we tested, the results have not been very conclusive, as seen, for instance, with *DBSCAN*. On the other hand, the tables show that the best results are observed in datasets with a larger quantity of data, such as hybrids and combustion cars. (Figure 5) This suggests that greater variety leads to better clustering performance, indicating that an appropriate clustering approach might involve using the complete dataset.

The method that produced the best results was *Agglomerative*, achieving a silhouette score of 0.95 for combustion cars and 0.69 for hybrids (Table 17), with 2 and 4 clusters, respectively. From this, we can conclude that these clusters are unrelated to price categories, which were used for external evaluation. Additionally, *OPTICS* yielded remarkable results.

From these findings, we can draw two conclusions. First, there is a clear hierarchy among the parameters, which aligns with the observation that tree-based algorithms performed the best across various tasks. (Tables 2-9) Second, the fact that *OPTICS* performed significantly better than *DBSCAN* suggests that the clusters have varying densities. This is consistent with the distribution of cars, which is highly skewed (Figure 5), and also explains why other methods did not perform well. Therefore, an appropriate method for clustering might be *HDBSCAN*.

Handling Best Hyperparameters

Each method has been tested with multiple hyperparameter combinations and optimized using Optuna.

The cuML library is leveraged to accelerate certain clustering algorithms, specifically `KMeans` and `DBSCAN`, by utilizing the computational power of the Graphics Processing Unit (GPU). This approach significantly improves the efficiency of clustering, especially with large datasets, compared to CPU-based implementations.

In the `ClusterGenerator` class, we check for the availability of cuML. If the library is available, we use the GPU-accelerated versions of the clustering algorithms: `cuKMeans` and `cuDBSCAN`. Otherwise, we fall back on the CPU-based versions from `sklearn`. These algorithms are integrated into the hyperparameter optimization process, where Optuna fine-tunes key parameters such as the number of clusters and initialization method.

Once the clustering models are trained, their performance is evaluated using several metrics: *silhouette_score*, *calinski_harabasz_score*, and *davies_bouldin_score*. If a model generates only a single cluster, it is penalized to avoid overfitting and to maintain a diverse set of clusters.

MasterGenerator Class

A `MasterGenerator` class that serves as a unified interface for generating and optimizing classification, regression, and clustering models has been implemented. It simplifies the process by automating model creation and optimization for all three tasks. The class includes methods for regression, classification, and clustering generation, using respective generators (`RegressionGenerator`, `ClassifierGenerator`, and `ClusterGenerator`). For more details, refer to the implementation on GitHub - MasterGenerator.

Classification

For each classification algorithm—Decision Tree, Random Forest, and SVM—an objective function is defined to maximize accuracy on a validation set. The hyperparameter search space is tailored to each method, focusing on factors like tree depth and splitting criteria for Decision Trees, the number of

estimators and leaf samples for Random Forest, and kernel types and regularization parameters for SVM. Once the best parameters are found, the models are retrained on the entire dataset to ensure optimal performance. This approach guarantees that the classifiers are fine-tuned to the specific characteristics of the data, resulting in more reliable and robust predictions.

The code related to the classification procedure can be found on GitHub - ClassifierGenerator

Regression

The RegressionGenerator class handles both the data management and optimization process for each regression method. An objective function is defined for each regression algorithm to minimize the Mean Absolute Percentage Error (MAPE) using a validation set.

For the Decision Tree Regressor, optimization includes parameters such as maximum depth, minimum samples split, minimum samples leaf, splitting criteria, and maximum features. For Random Forest, we optimize the number of estimators, maximum depth, minimum samples split, and bootstrap sampling. Both sklearn and cuML implementations are available, depending on the system capabilities. For SVR, key parameters like kernel type, regularization parameter (C), epsilon, degree for polynomial kernels, and gamma are adjusted. cuML and sklearn implementations are used for linear and other kernel types, respectively. Linear Regression, having no hyperparameters to tune, is applied directly.

A predefined number of trials are run for each algorithm to determine the best hyperparameters. After optimization, the model is retrained on the full dataset using the identified parameters to ensure consistent and robust predictions. The results, including the best hyperparameters and performance metrics such as MAPE, are stored for further analysis. The Evaluator class also provides visual and numerical summaries of the regression performance.

The code related to the regression procedure can be found on GitHub - RegressorGenerator

Clustering

For clustering algorithms—K-Means, Agglomerative Clustering, DBSCAN, Birch, OPTICS, and Gaussian Mixture Model (GMM)—the objective function is designed to maximize the silhouette score, ensuring the best possible separation between clusters. The hyperparameter search space is customized for each algorithm, focusing on factors like the number of clusters for K-Means and Agglomerative Clustering, epsilon and minimum samples for DBSCAN, and covariance types for GMM.

To evaluate the performance of each clustering method, both internal metrics (such as silhouette score, Calinski-Harabasz index, and Davies-Bouldin index) and external metrics (like Adjusted Rand Index, Normalized Mutual Information, and Homogeneity) are computed when applicable. This evaluation guarantees that the clustering process is reliable and enables meaningful comparisons between the algorithms.

The code related to the clustering procedure can be found on GitHub - ClusterGenerator

Development Report

To conclude, our workflow consisted of the following steps: first, collecting data in JSON format, followed by processing it into a series of CSV files. Subsequently, we tested the models taught in class, focusing on regression and classification using default parameters. After partially analyzing the information, we fine-tuned the models with hyperparameters and applied clustering techniques to extract insights from the texts. For instance, among the various attributes of a second-hand car, we identified that some features have a greater influence on the price than others.

During the hyperparameter tuning process, we encountered challenges related to time and computational capacity, as many configurations exceeded our available memory. To address these issues, we implemented techniques such as PCA or TruncatedSVD to reduce the dimensionality of our data and leveraged libraries that support GPU acceleration.

Despite these efforts, we believe that with greater computational capacity, it would have been possible to achieve more refined and accurate models.

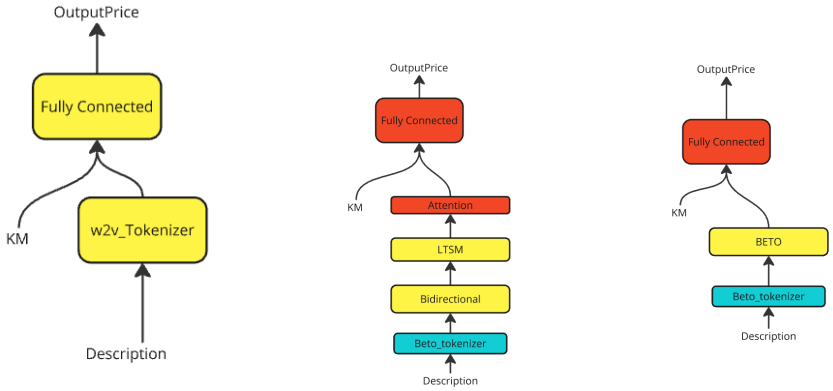
Natural Language Processing (Bonus task)

Algorithm	MAE	MSE	MAPE	R2	Error Mean	Error Std Dev	Adjusted R2
Fully Connected	2.6275×10^3	8.7585×10^7	1.3690×10^1	0.8126	2.6275×10^3	8.9823×10^3	0.8125
LTSM BI	2.5556×10^3	2.9960×10^7	1.3383×10^1	0.9346	3.5564×10^3	6.2622×10^3	0.8868
BETO	3.5564×10^3	5.1863×10^7	1.8221×10^1	0.8868	3.5564×10^3	6.2622×10^3	0.8868

Table 22: Comparison of algorithms based on MAE, MSE, MAPE, R2, Error Mean, Error Std Dev, and Adjusted R2.

Objective

The objective of this study is to use the text appearing in the descriptions of cars, along with their mileage, to create a model capable of predicting the price of a car. For this purpose, three models of varying complexities have been developed.



(a) Prototype1 (b) Prototype2 (c) Prototype3

Figure 19: NLP models

First Prototype

The first prototype (Figure 19.a), characterized by a simple structure, provides commendable results given its simplicity. However, it lacks robustness, as altering the order of phrases in the descriptions significantly changes the output. This is due to the model's reliance solely on tokenizing the text, effectively classifying text types into different price ranges rather than extracting meaningful semantic relationships. It is worth noting that the computational cost for training this model was very low, taking approximately 30 minutes.

Second Prototype

The second prototype (Figure 19.b), slightly more complex, first tokenizes the text using BETO, a pre-trained model for Spanish text, and then processes the tokens through a bidirectional architecture. This enables the model to account for changes in phrase order. Additionally, it incorporates an LSTM layer and an attention mechanism. This model demonstrates greater robustness and improved predictions for outlier cases compared to the first prototype. Although the computational cost is significantly higher, the loss curves indicate that the model could be further improved with more data. The presence of local minima suggests that the computational resources available were insufficient to achieve optimal performance (Figure 20.b).

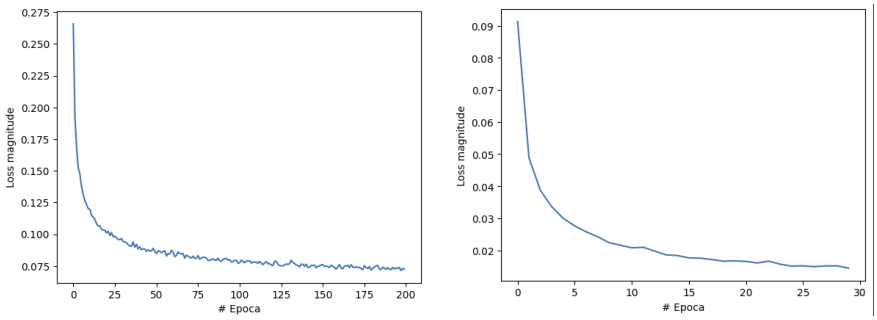
Third Model

The third model (Figure 19.c) employs a pre-trained BETO model specifically designed for Spanish text. This model exhibits the highest capacity among the three, as it tolerates human-written text, even though such text is outside the domain of this task since most descriptions are programmatically generated. The computational resources required to train this model are substantially greater than those of the other two. While its prediction accuracy leaves room for improvement, it performs significantly better than the first prototype for extreme cases.

Analysis of Text Generation and Model Suitability

If the texts had been generated naturally, i.e., manually introduced by the car sellers, the third model, which leverages a pre-trained model for Spanish texts to predict prices, would have been the most appropriate choice. However, since the predictions are based on systematically generated texts, albeit highly complex with numerous variations, the second model emerges as the most suitable for this task.

Despite its lower capacity and inability to process paraphrased texts effectively, the second model is well-suited because the generated texts consist of fixed phrases presented in different orders. This characteristic aligns perfectly with the model's capabilities. Using this model, we can predict, based on a description and mileage, whether a car's price significantly exceeds the predicted price. Such discrepancies would likely indicate an overpriced listing.



(a) Prototype1 (b) Prototype2

Figure 20: NLP models

Conclusion

We conclude that it is not always necessary to select the most complex models for simple tasks. Instead, identifying the model most appropriate for the task at hand can maximize efficiency in resource and time utilization. This approach highlights the importance of aligning model complexity with task requirements to achieve optimal performance.