Car Price Prediction from Mobile.de

Motivation

There are few things in life which us, Romanians, love more than german cars. But buying one which has already been imported into our country can be a challenge. Luckily for us, mobile.de is the biggest car sales platform in all of Europe, and what better place to buy german cars from then...well, Germany?

However, the automotive market is dynamic, with car prices influenced by numerous factors such as make, model, age, mileage, and location. Understanding and predicting car prices can help buyers make informed decisions and sellers optimize their listings. This project aims to leverage data from Mobile.de, to build a machine learning model that predicts car prices accurately.

Dataset Creation

Web Scraping Process

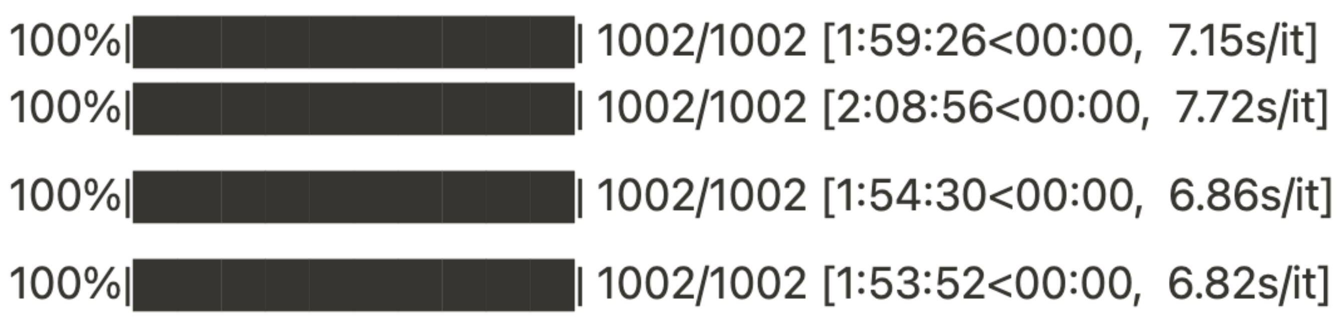
To build a robust dataset for car price prediction, data was scraped from Mobile.de. This process involved:

1. **Setup**:
    - Libraries such as `selenium` and `BeautifulSoup` were used for web scraping, enabling automated browsing and HTML parsing.
    - `webdriver-manager` was used for the setup of Chrome WebDriver.
2. **Target Selection**:
    - The dataset focused on cars from specific manufacturers (the VAG Group, consisting of Audi, Volkswagen, Skoda, Seat).
    - Each manufacturer was identified by a unique ID provided by Mobile.de.
3. **Scraping Workflow**:
    - **Page Navigation**:
        - A `scrape_page_for_make` function was created to navigate to car listings for a given manufacturer and page number.
        - Cookie consent pop-ups were handled programmatically to ensure smooth scraping.
    - **Data Extraction**:
        - HTML content was fetched and parsed using `BeautifulSoup`.
        - A dedicated function, `extract_listing_links_from_soup`, extracted car listing links from the parsed content.
4. **Iterative Scraping**:
    - The script iterated through up to 50 pages for each manufacturer, collecting listing URLs.
    - Scraping was paused briefly between requests to avoid detection and ensure compliance with website policies.
5. **Data Storage**:
    - Extracted links were stored in a dictionary, organized by manufacturer, for further processing

```
def scrape_page_for_make(make, page_number):
    # Create a new browser instance
    driver = webdriver.Chrome(service=ChromeService(ChromeDriverManager().install()))
    # Create the base url
    link = f"https://suchen.mobile.de/fahrzeuge/search.html?dam=false&isSearchRequest=true&p=%3A

    if page_number > 1:
        link += f"&pageNumber={page_number}"
    print(f"Starting to scrape {link}")

    try:
        driver.get(link)

        # Wait for the page to load
        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.TAG_NAME, "body"))
        )

        # Try to handle the cookie consent pop-up
        try:
            cookie_button = WebDriverWait(driver, 5).until(
                EC.element_to_be_clickable(
                    (By.XPATH, "//button[contains(text(), 'Accept')]")
                )
            )
            cookie_button.click()
            print("Cookie consent accepted.")
        except Exception as e:
            print("No cookie consent pop-up or unable to locate it:", e)

        # Pause to let the page fully load
        time.sleep(1)

        listings_source = driver.page_source
        listings_soup = BeautifulSoup(listings_source, "html.parser")
        print("Done scraping base link")

    finally:
        # Always close the driver
        driver.quit()

    return listings_soup
```

Dataset Creation Outcome

- After (many) hours of webscraping, the dataset was complete. Consisting of 4000 car listings with prices ranging from €5000 to €30000.
- After converting the dictionary to a Pandas DataFrame, we noticed we gathered 48 features for the listings, many of which will need cleaning



```
100%|██████████| 1002/1002 [1:59:26<00:00,  7.15s/it]
100%|██████████| 1002/1002 [2:08:56<00:00,  7.72s/it]
100%|██████████| 1002/1002 [1:54:30<00:00,  6.86s/it]
100%|██████████| 1002/1002 [1:53:52<00:00,  6.82s/it]
```

Webscraping Process Progress

Exploratory Data Analysis

Data Overview

The dataset contains detailed information on cars, including variables such as:

- **Price**: The asking price of the car (in Euros).
- **Mileage**: Distance driven by the car (in kilometers).
- **Power**: Engine power (in kilowatts).

- **Cubic Capacity**: Engine displacement (in cubic centimeters).
- **Make**: The car manufacturer (e.g., Audi, Volkswagen).
- Additional features such as `Fuel Type`, `Transmission`, and `Year of Manufacture`.

All features of the dataset:

Column Name	Non-null Count	Type
Vehicle condition	4005	object
Category	4005	object
Vehicle Number	2320	object
Availability	1665	object
Origin	2777	object
Mileage	4005	object
Cubic Capacity	3996	object
Power	4005	object
Drive type	4005	object
Fuel	4005	object
Number of Seats	3942	object
Door Count	3995	object
Transmission	4001	object
Emission Class	3817	object
Emissions Sticker	3743	object
First Registration	4005	object
HU	3689	object
Climatisation	3991	object
Parking sensors	3906	object
Airbags	3897	object
Colour (Manufacturer)	3701	object
Colour	3981	object
Interior Design	3979	object
price	4005	object
Make	4005	object
Energy consumption (comb.)2	1995	object
CO_2 emissions (comb.)2	1995	object
Fuel consumption2	1936	object
Number of Vehicle Owners	2989	object
Trailer load braked	1374	object
Trailer load unbraked	1360	object
Weight	1844	object
Cylinders	2468	object
Tank capacity	1387	object
Date of last service (date)	412	object
Last service (mileage)	425	object
CO_2 class	351	object
Energy costs for 15,000 km annual mileage2	184	object
Fuel price	148	object
Vehicle tax	162	object
Construction Year	379	object
Support load	97	object

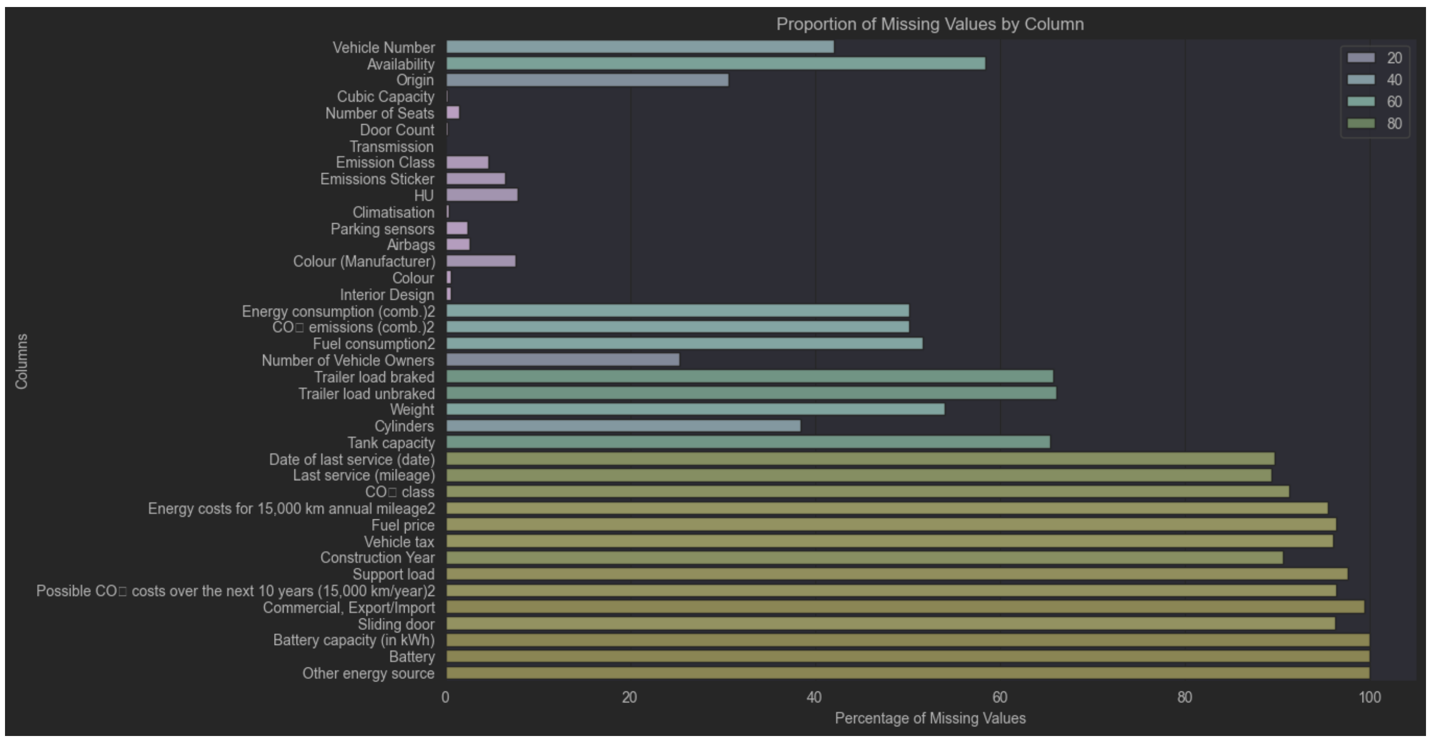
Possible CO_2 costs over the next 10 years (15,000 km/year)2	147	object
Commercial, Export/Import	24	object
Sliding door	151	object
Battery capacity (in kWh)	3	object
Battery	1	object
Other energy source	1	object

Data Cleaning For Numeric Features

- We notice that all columns of the dataset have a type of 'object' meaning they are stored as strings. However, many of these columns are actually numeric, so we need to convert them to their actual data type.

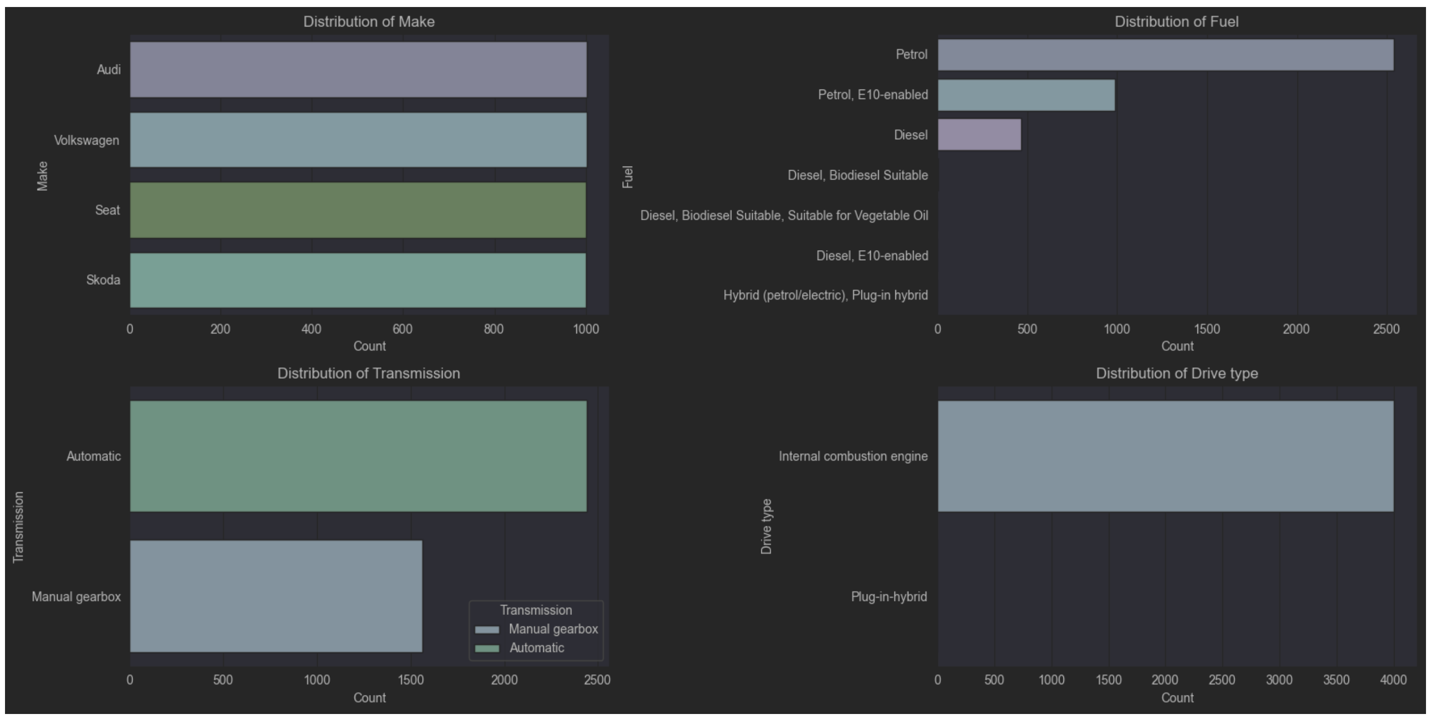
```
# Function to clean numeric columns
def clean_numeric_column(column, remove_text=True):
    if remove_text:
        return column.str.replace(r'[^\d.]', '', regex=True).astype(float)
    return column

# Convert numeric-like columns to numeric
data_df['price'] = clean_numeric_column(data_df['price'])
data_df['Mileage'] = clean_numeric_column(data_df['Mileage'])
data_df['Cubic Capacity'] = clean_numeric_column(data_df['Cubic Capacity'])
data_df['Power'] = data_df['Power'].str.extract(r'(\d+)', expand=False).astype(float)
```

- Another thing to note is that many columns contain null values, which we can visualize. To fix this, we will drop all columns which have more than 50% missing values in them.



Categorical Features

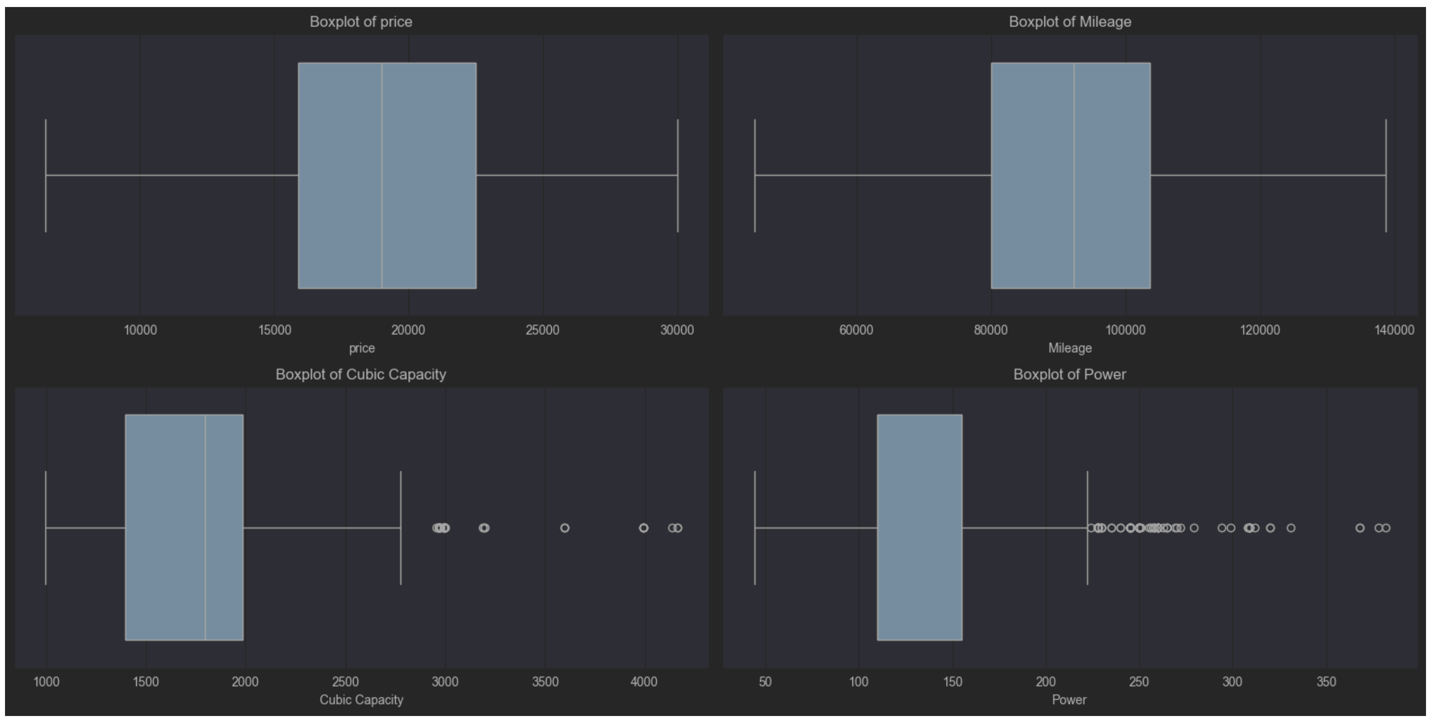
- With the numeric features out of the way, we can take a quick look at the categorical ones, more specifically, their distribution



- We notice the makes are well distributed across the dataset, which is to be expected.
- Another thing to be expected, since the price range caps of at 30000, is that almost all car listings only have an internal combustion engine, with a few having a plug-in-hybrid system.
- Since Diesel engines are becoming less and less popular, we notice the majority of the listings have a Petrol fueled engine instead.
- And lastly, we can see a preference towards automatic transmissions instead of manual ones.

Outlier Detection

- We can visualize the outliers in our most important numeric columns using boxplots.

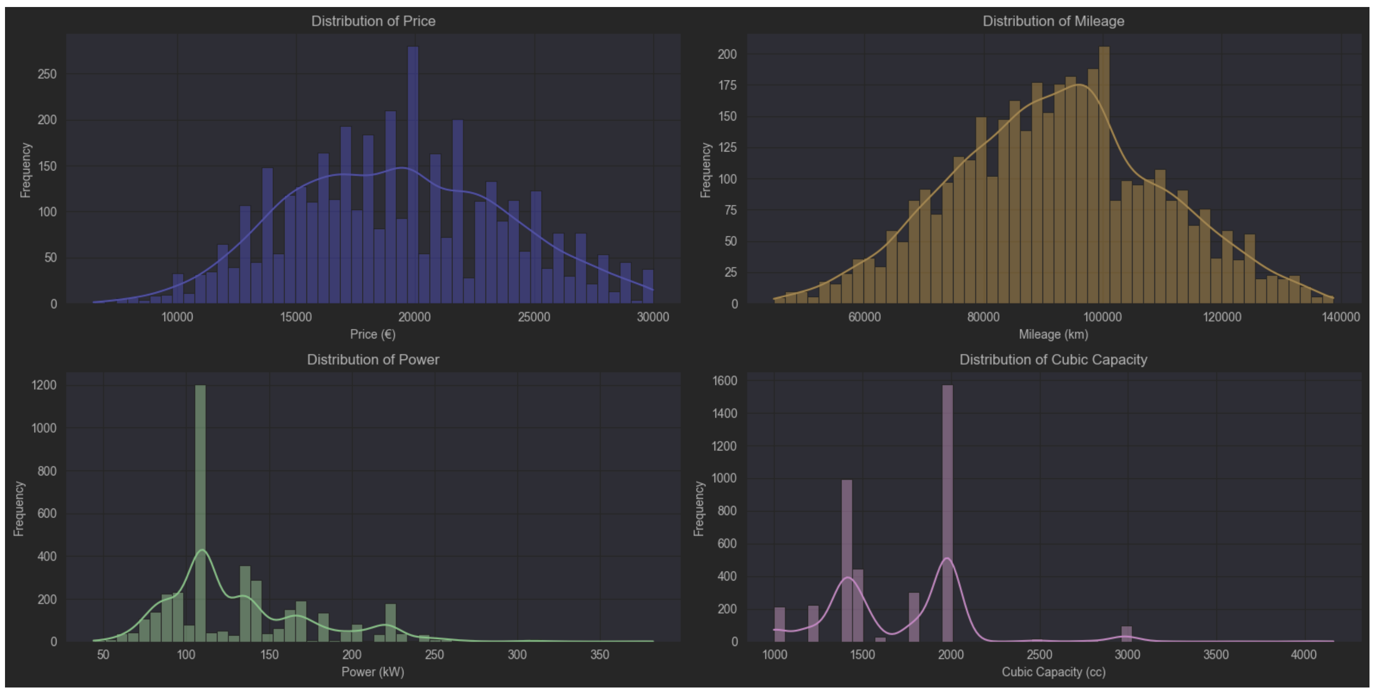


- To handle these outliers, we can use the IQR method:

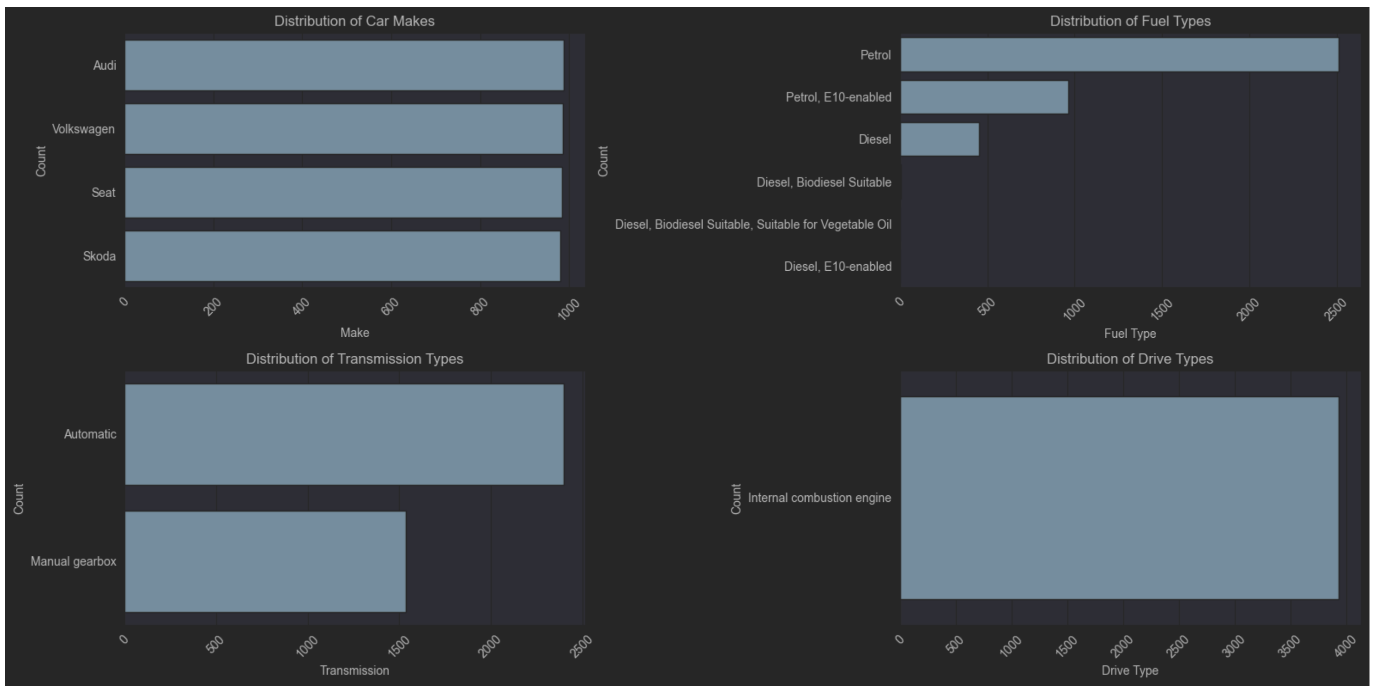
```
for col in ['price', 'Mileage']:
    Q1 = data_df_cleaned[col].quantile(0.25)
    Q3 = data_df_cleaned[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    data_df_cleaned = data_df_cleaned[(data_df_cleaned[col] >= lower_bound) & (data_df_cleane
```

Univariate Analysis

- We can visualize the distribution of our key numerical and categorical features to better understand them



Distribution of Key Numeric Features



Distribution of Key Categorical Features

Conclusions from Univariate Analysis

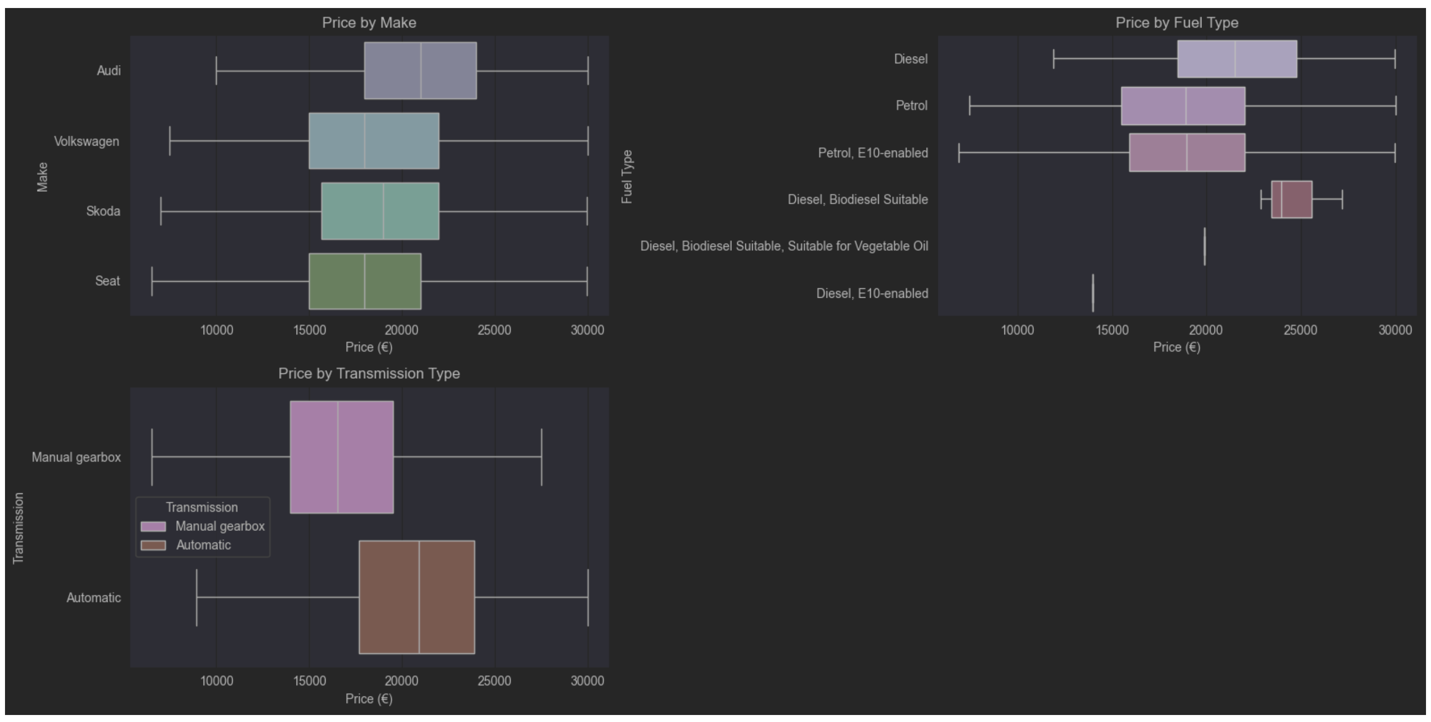
1. Price
    - The price distribution is normal, with most vehicles priced around €20,000.
    - A small number of luxury or high-end vehicles create outliers at the higher end.
2. Mileage
    - Mileage follows a somewhat bell-shaped curve, with most vehicles between 50,000 km and 140,000 km.
    - Few vehicles have extremely low or high mileage, but they may represent unique cases (e.g., new or heavily used vehicles).
3. Power
    - The power distribution shows a clustering around 100–200 kW, which is common for German vehicles.
    - Vehicles with very high power are rare, likely reflecting high-performance or specialty cars.
4. Cubic Capacity
    - The cubic capacity distribution peaks around 1,500–2,000 cc, which aligns with typical engine sizes for German cars.
    - Larger engines (e.g., above 3,000 cc) are less frequent and likely associated with premium or performance cars.

Bivariate Analysis

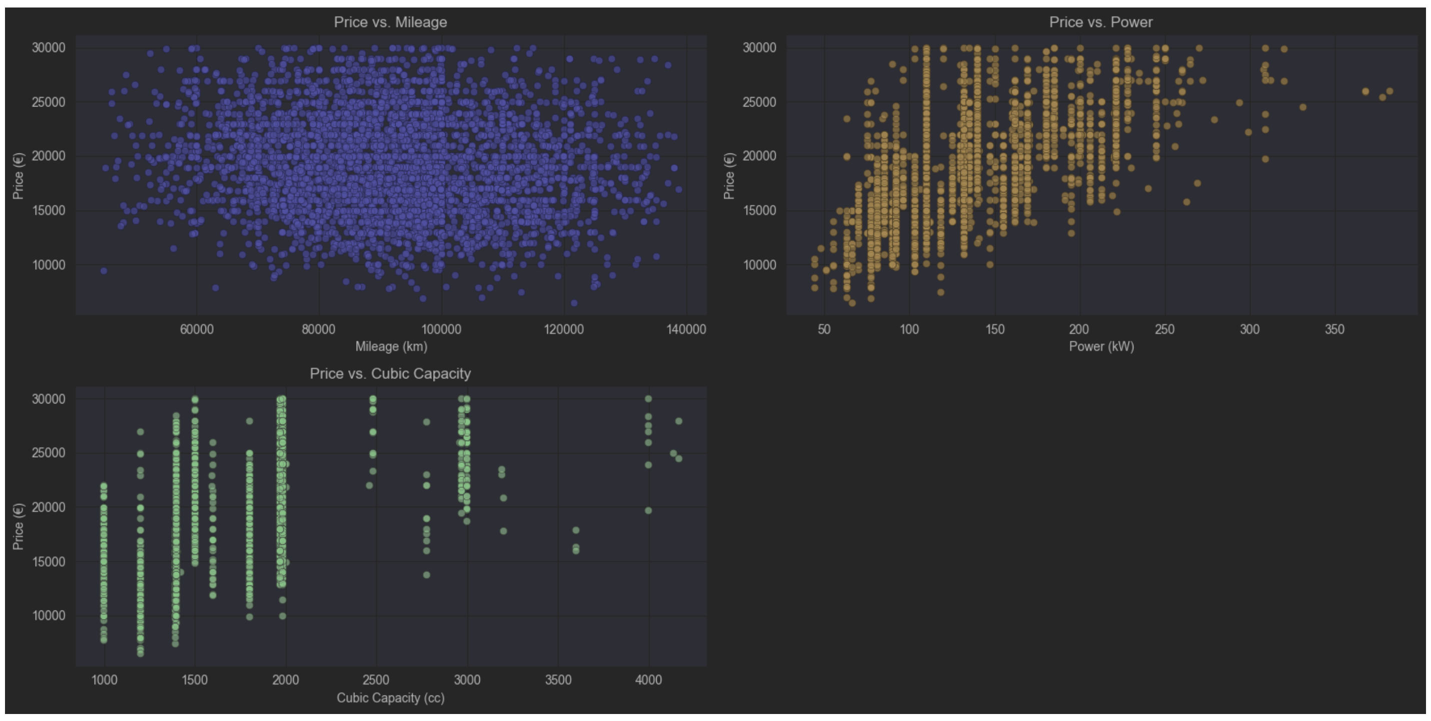
Purpose:

To explore relationships between pairs of variables, such as:

- price vs. Mileage
- price vs. Power
- price vs. categorical variables like Make or Fuel



Bivariate analysis of Price vs Key Categorical Features

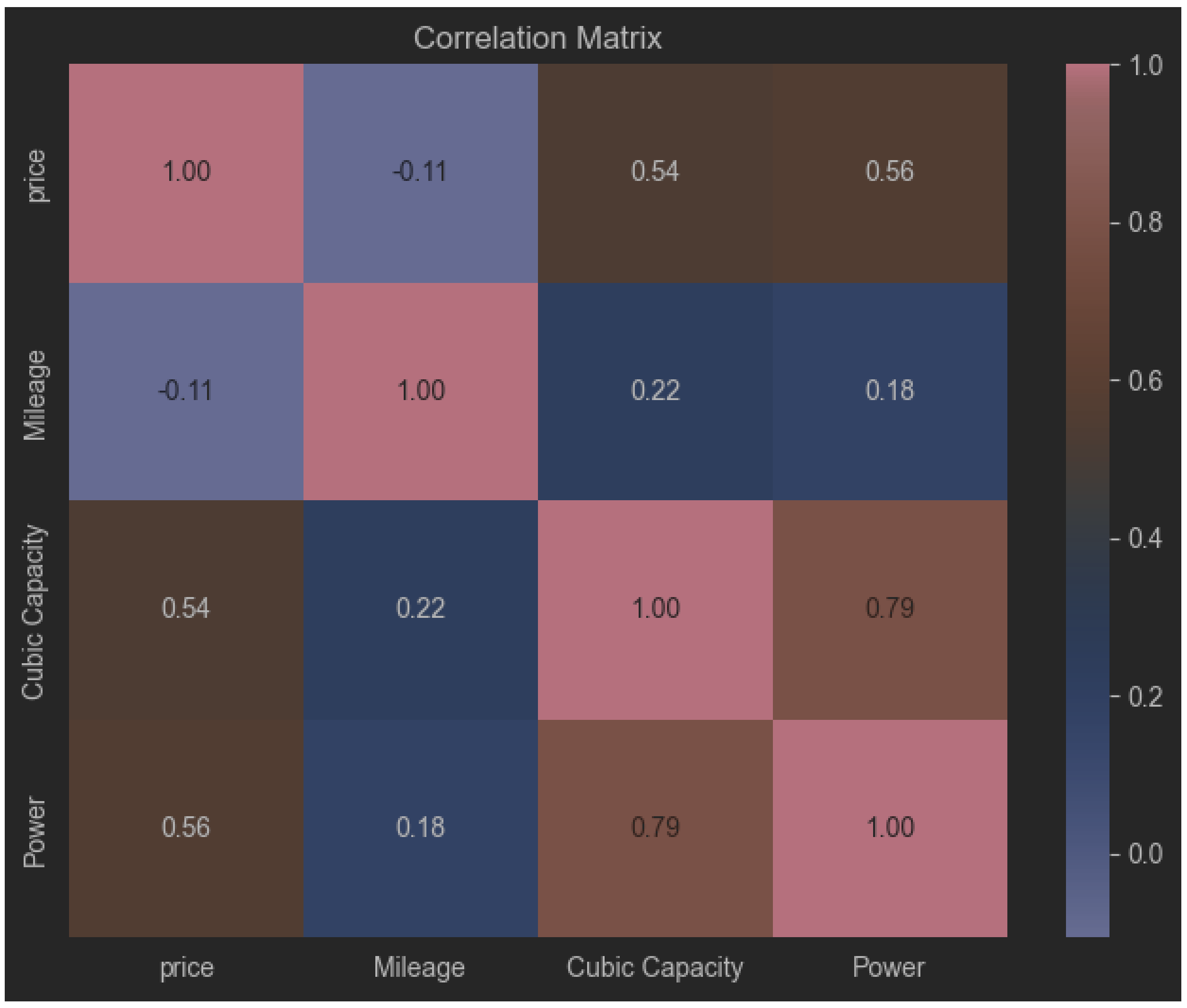


Bivariate analysis of Price vs Key Numeric Features

Correlation Analysis

Purpose:

To identify linear relationships between numeric variables and the target variable (price).

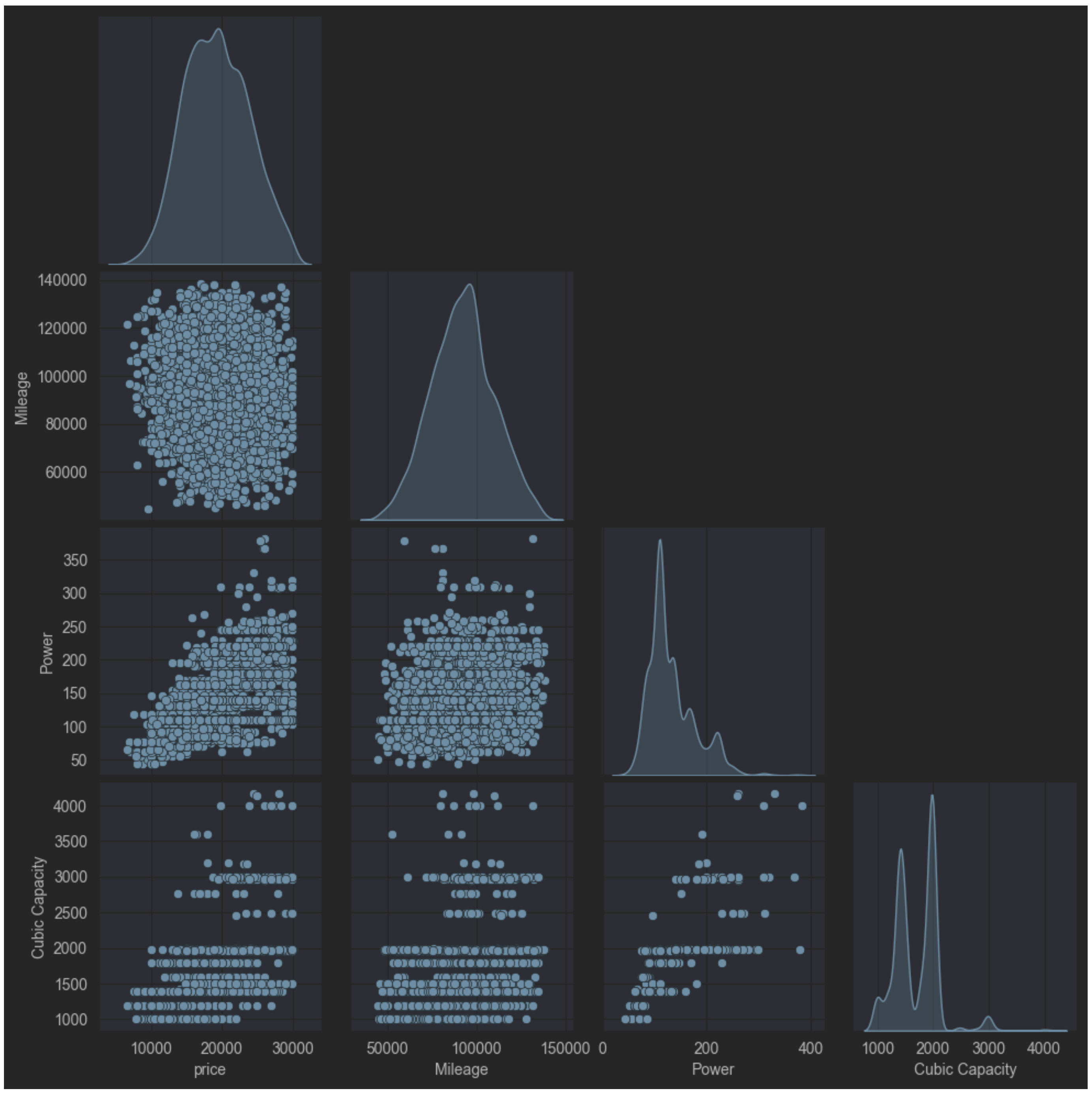


Correlation Matrix

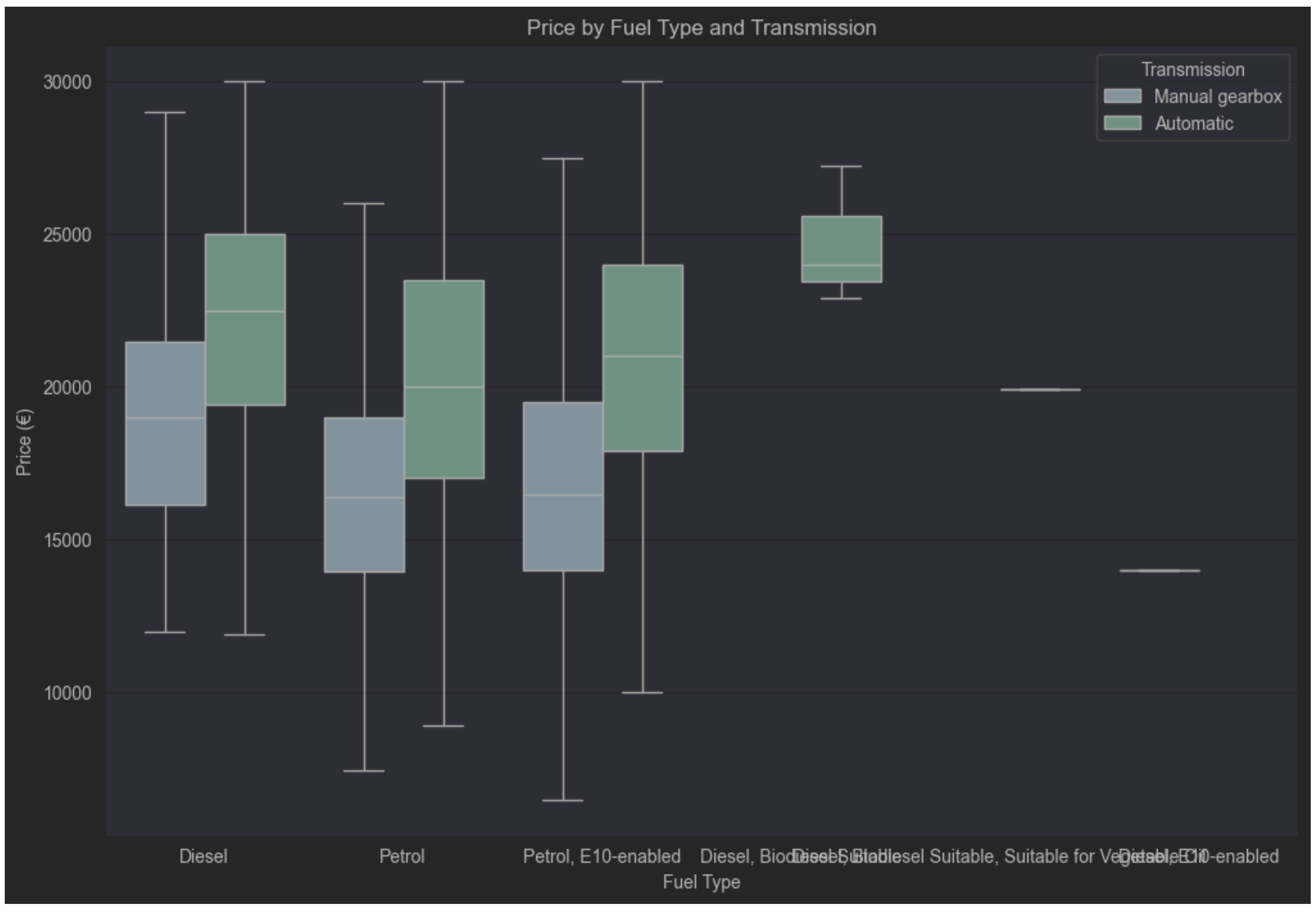
Multivariate Analysis

Purpose:

Explore how multiple variables interact, e.g., the effect of Make and Fuel type on price.



Pair Plot



Box Plot With Two Categorical Features

Analysis Conclusions

- Bivariate Analysis
  - Price vs. Mileage:
    - Clear negative relationship: cars with higher mileage generally have lower prices.

        - A few outliers exist (e.g., low-price cars with very low mileage).
    - Price vs. Power:
        - Positive relationship: cars with higher power (kW) tend to have higher prices.
        - The trend weakens for vehicles with extremely high power.
    - Price vs. Cubic Capacity:
        - Moderate positive relationship: larger engines generally correlate with higher prices.
        - A notable clustering around common cubic capacities (e.g., ~2,000 cc).
- Price by Categorical Features
    - Price by Make:
        - Luxury brands like Audi and BMW have higher median prices compared to Skoda and Volkswagen.
        - Price variance is highest for Audi, reflecting its diverse product range.
    - Price by Fuel Type:
        - Diesel cars tend to have slightly higher prices compared to petrol cars.
        - Alternative fuels (e.g., hybrid) appear less frequently but with higher prices.
    - Price by Transmission:
        - Automatic vehicles have higher median prices compared to manual ones.
        - Likely reflects demand and technology differences.
- Correlation Analysis
    - Strongest Correlations:
        - Power has a moderately strong positive correlation with price (~0.6).
        - Mileage has a moderate negative correlation with price (~-0.5).
        - Cubic Capacity shows a weaker positive correlation with price (~0.5).
    - Key Insights:
        - Engine-related metrics (Power, Cubic Capacity) and Mileage are important predictors of price.
        - Multicollinearity among numeric variables appears limited, which is ideal for modeling.

Feature Engineering

We engineered several new features to improve our model:

- Derived Features
    - Vehicle Age: Created from the First Registration date
    - Mileage per Year: Calculated by dividing mileage by vehicle age
- Encoded Features
    - We converted categorical columns like Make, Fuel, Transmission, and Drive type into numerical format using one-hot encoding
    - We removed less relevant categorical columns with too many unique values (like Interior Color)
- Transformed Features
    - We applied log transformations to price and mileage to handle their skewed distributions
- Standardized Features
    - We standardized all key numeric features (Mileage, Power, Cubic Capacity, Vehicle Age, Mileage per Year) to put them on the same scale

```
# Step 1: Derive Features
current_year = datetime.now().year
```

```
data_df_cleaned['First Registration Year'] = data_df_cleaned['First Registration'].str.extract(r
data_df_cleaned['Vehicle Age'] = current_year - data_df_cleaned['First Registration Year']
data_df_cleaned['Mileage per Year'] = data_df_cleaned['Mileage'] / data_df_cleaned['Vehicle Age

# Step 2: Encoding All Categorical Variables
# Select columns for one-hot encoding
categorical_columns_to_encode = ['Vehicle condition', 'Make', 'Category', 'Door Count', 'Emissic

# Perform one-hot encoding
encoder = OneHotEncoder(sparse_output=False)
encoded_columns = pd.DataFrame(
    encoder.fit_transform(data_df_cleaned[categorical_columns_to_encode]),
    columns=encoder.get_feature_names_out(categorical_columns_to_encode),
    index=data_df_cleaned.index
)

# Drop all remaining categorical columns
categorical_columns_to_drop = data_df_cleaned.select_dtypes(include='object').columns
data_df_encoded = pd.concat([data_df_cleaned.drop(columns=categorical_columns_to_drop), encoded_

# Step 3: Handle Skewness
data_df_encoded['Log Price'] = np.log1p(data_df_encoded['price'])
data_df_encoded['Log Mileage'] = np.log1p(data_df_encoded['Mileage'])

# Step 4: Standardization
numeric_features = ['Mileage', 'Power', 'Cubic Capacity', 'Vehicle Age', 'Mileage per Year']
scaler = StandardScaler()
data_df_encoded[numeric_features] = scaler.fit_transform(data_df_encoded[numeric_features])
```

Train/Test Split

- The first step in building our models will be splitting the data into a train set and a test set. We use an 80/20 ratio.
- We will also log-transform the target variable (the price).

```
# Split the data
X = data_df_encoded.drop(columns=['price', 'Log Price'])  # Features
y = data_df_encoded['Log Price']  # Target (log-transformed price)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Feature Selection

- Since after one-hot-encoding our dataset now consists of over 140 features, many of them being highly correlated with each other, we need to select the best ones for our model. We do this in 3 steps:
  - First, we remove features which have a correlation of over 85%
  - Then, we train a basic model, say a Random Forest Model, sort all features by their importance evaluated by the model. We then select the top 25 most important features.
  - Finally, we perform Recursive Feature Elimination (RFE), again using a Random Forest Model to only select the 20 most important features.

  ```
  # Step 3: Feature Selection

  # (a) Correlation-Based Filtering
  correlation_matrix = X.corr().abs()
  upper_triangle = correlation_matrix.where(np.triu(np.ones(correlation_matrix.shape), k=1).ast
  high_correlation_features = [column for column in upper_triangle.columns if any(upper_triang]
  ```

```
X_selected = X.drop(columns=high_correlation_features)

# Subset X_selected to match train indices
X_train_selected = X_train[X_selected.columns]

# (b) Univariate Feature Importance
# Use RandomForest to calculate feature importances
importance_model = RandomForestRegressor(random_state=42)
importance_model.fit(X_train, y_train)

# Rank features by importance
feature_importances = pd.Series(importance_model.feature_importances_, index=X_train.columns
top_features = feature_importances.nlargest(25).index

# Ensure top features exist in X_train_selected
top_features = [feature for feature in top_features if feature in X_train_selected.columns]
X_train_selected = X_train_selected[top_features]

# (c) Recursive Feature Elimination (RFE)
rfe_model = RandomForestRegressor(random_state=42)
rfe = RFE(estimator=rfe_model, n_features_to_select=20, step=1)

# Fit RFE on the subset of X_train_selected and y_train
rfe.fit(X_train_selected, y_train)

# Select final features based on RFE
selected_features_rfe = X_train_selected.columns[rfe.support_]
X_train_final = X_train_selected[selected_features_rfe]

# Apply the same transformation to X_test
X_test_final = X_test[X_train_final.columns]

print("Selected Features After RFE:", list(selected_features_rfe))
```

Selected Features After Elimination

- Cubic Capacity, Power, First Registration Year, Category Van/Minibus, Make Audi, Mileage, Emission Class Euro5, Mileage per Year, Make Seat, Category SUV/Off-road Vehicle/Pickup Truck, Transmission Automatic, Climatisation Automatic (3 zones), Make Volkswagen, Make Skoda, Climatisation Automatic (2 zones), Category Saloon, Parking Sensors Rear, Fuel Petrol, Parking Sensors Front and Rear, Vehicle Condition Used

Baseline Model Training

We chose to train 4 types of regression models to predict the price of the cars. These include:

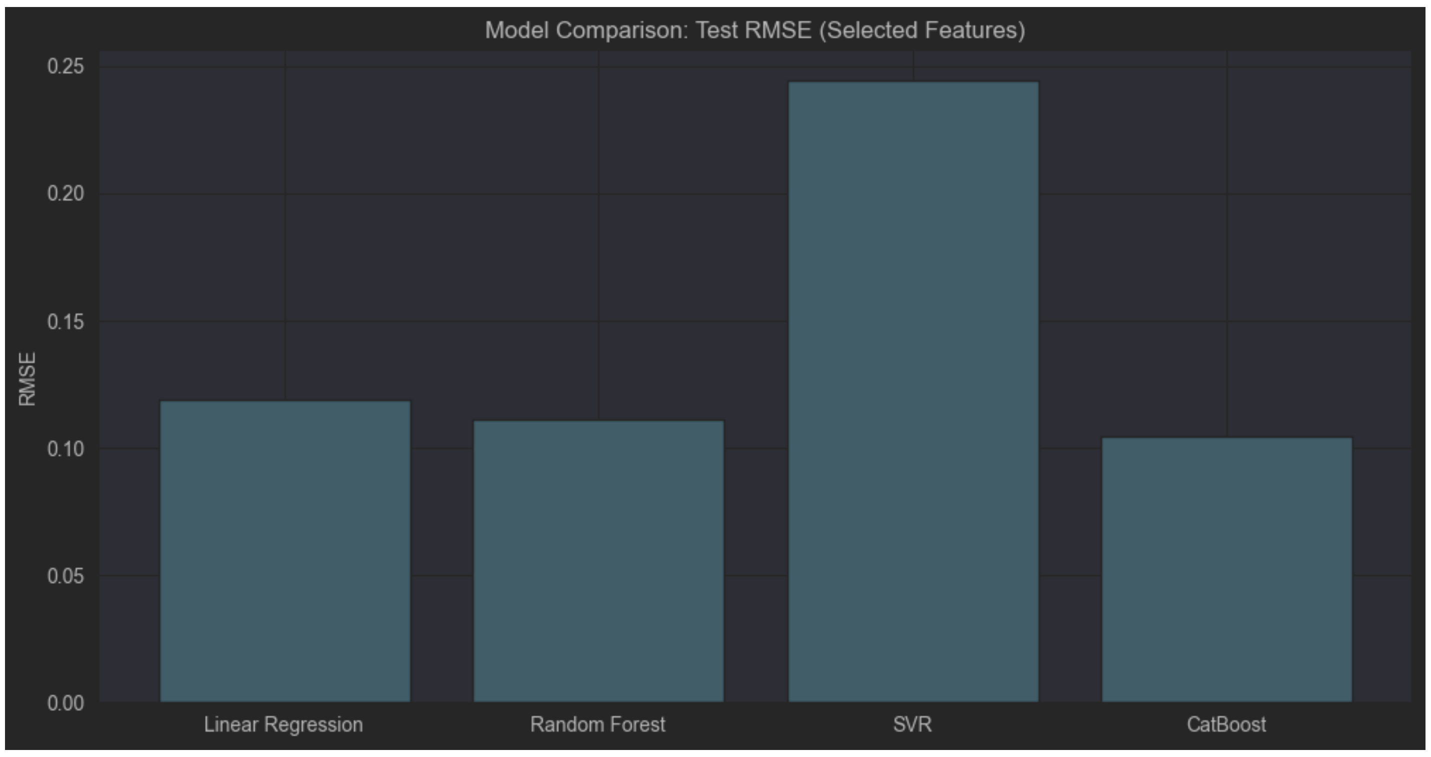
- Linear Regression
- Random Forest Regressor
- SVR (Support Vector Regression)
- CatBoostRegressor (Categorical Gradient Boosting)

First, we train the models using set hyperparameters to get a baseline for their performance. We then evaluate their Root Mean Squared Error and their R².

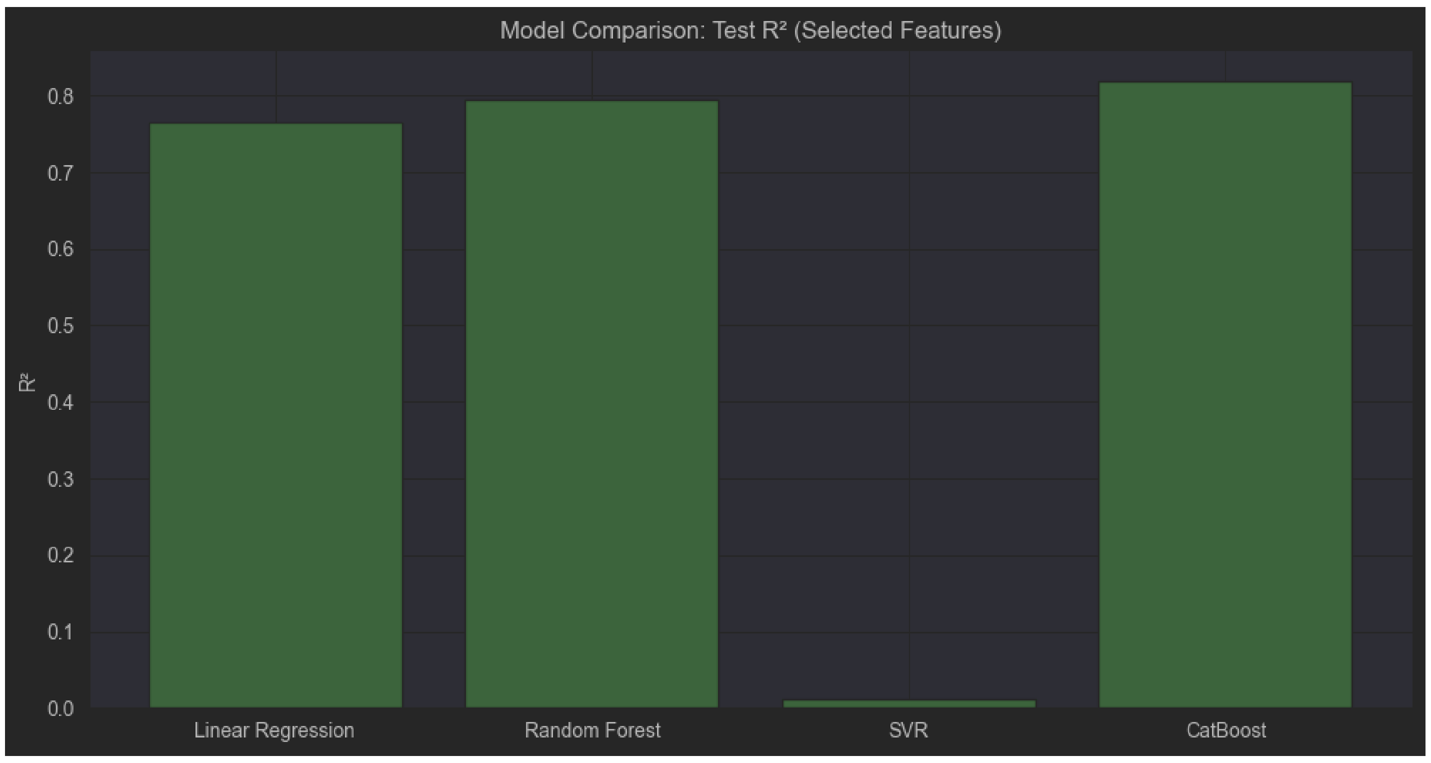
```
# Train Models (Without Hyperparameter Tuning) with Selected Features to get a sense of perfo
models = {
    "Linear Regression": LinearRegression(),
    "Random Forest": RandomForestRegressor(random_state=42, max_depth=10, n_estimators=100),
    "SVR": SVR(kernel='rbf'),
    "CatBoost": CatBoostRegressor(verbose=0, random_state=42, learning_rate=0.05, iterations=
}

baseline_results = {}
for name, model in models.items():
    model.fit(X_train_final, y_train)

    predictions_test = model.predict(X_test_final)
    predictions_train = model.predict(X_train_final)

    rmse_test = np.sqrt(mean_squared_error(y_test, predictions_test))
    r2_test= r2_score(y_test, predictions_test)

    rmse_train = np.sqrt(mean_squared_error(y_train, predictions_train))
    r2_train = r2_score(y_train, predictions_train)

    baseline_results[name] = {'RMSE Test': rmse_test, 'R² Test': r2_test, 'RMSE Train': rmse_

print("Baseline Model Results:")
for name, metrics in baseline_results.items():
    print(f"{name}: RMSE Train = {metrics['RMSE Train']:.4f}, R² Train = {metrics['R² Train']
          f"RMSE Test = {metrics['RMSE Test']:.4f}, R² Test = {metrics['R² Test']:.4f}")
```

Baseline Models Results

Model	RMSE Train	R^2 Train	RMSE Test	R^2 Test
Linear Regression	0.1191	0.7736	0.1190	0.7651
Random Forest	0.0668	0.9287	0.1113	0.7945
SVR	0.2488	0.0122	0.2441	0.0124
CatBoost	0.0866	0.8803	0.1047	0.8184



Baseline Test RMSE



Baseline Test R²

Baseline Models Evaluation

- The Linear Regression Model performs quite well, considering how simple it is and does not overfit
- The Random Forest Model performs the best on the training data but overfits quite a bit, offering worse performance on the test data
- The Support Vector Regression Model performs very poorly on both the training and testing data and will definitely need hyperparameter tuning
- Lastle, the Catboost Model offers the best performance on the testing dataset while overfitting less then the Random Forest Model

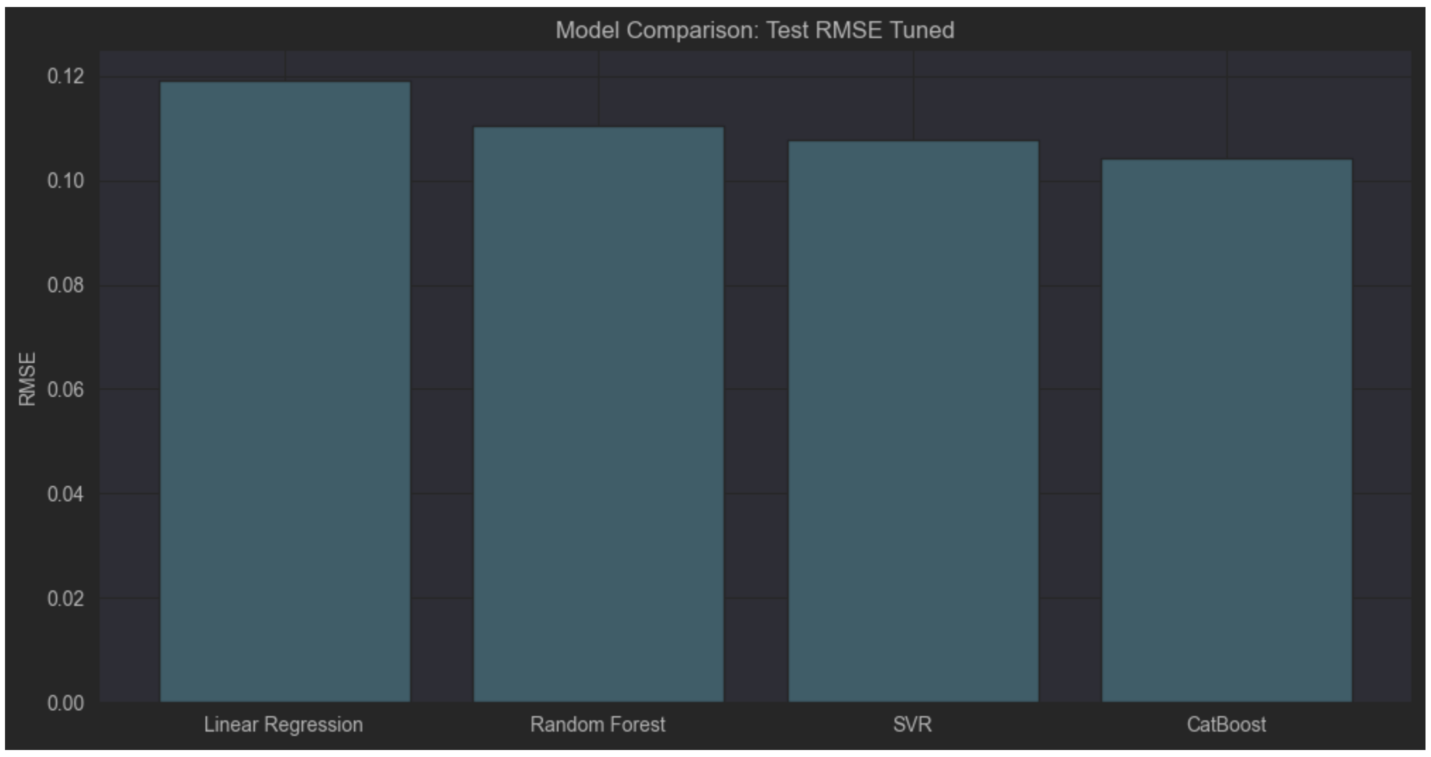
Hyperparameter Tuning

To get better performance out of our models, we need to tune their hyperparameters, especially for the SVM model. We do this by performing a grid search over the hyperparameters of each model and selecting the best ones using Cross Validation.

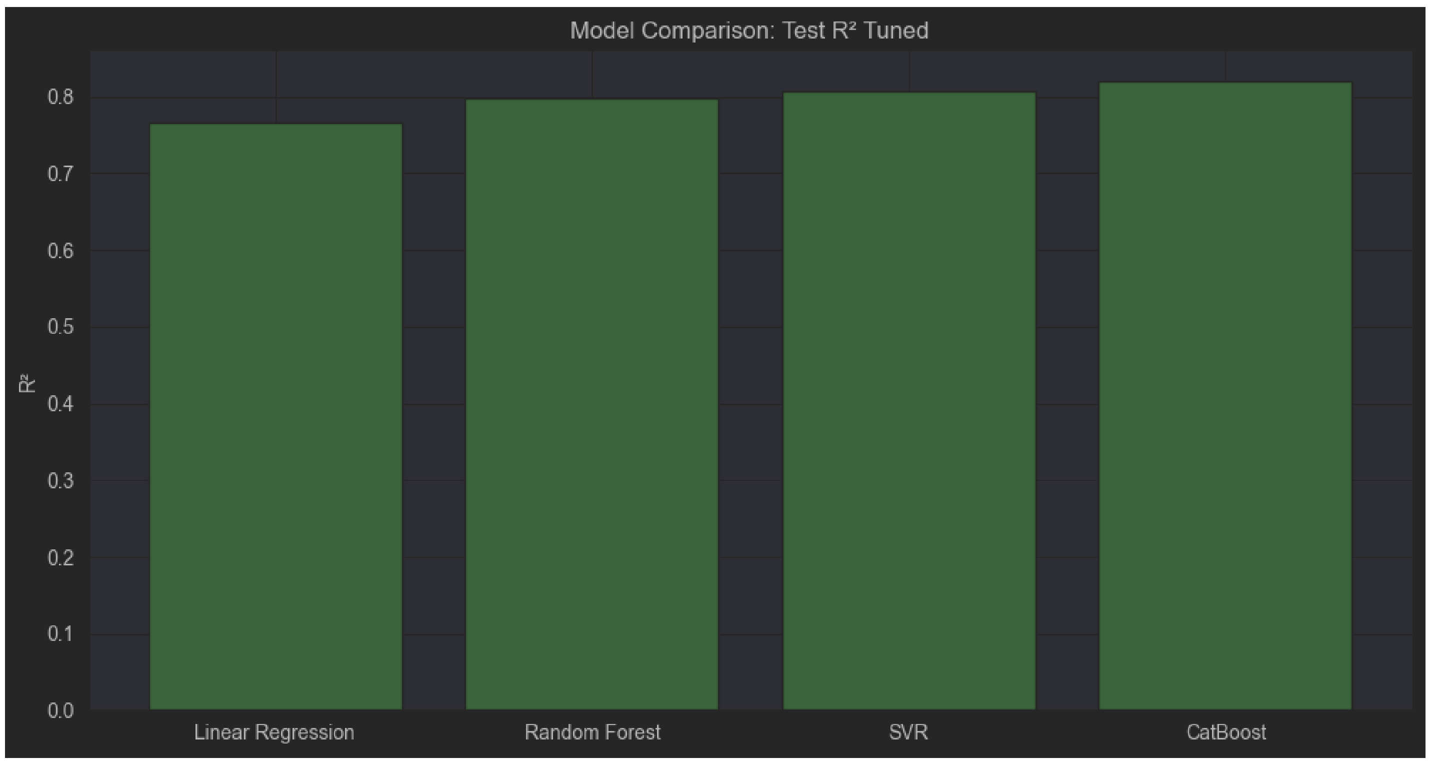
```
# Re-train Models with Hyperparameter Tuning

# Define hyperparameter grids for tuning
param_grids = {
    'Linear Regression': {},
    'SVR': {'C': [0.1, 1, 10], 'gamma': ['scale', 'auto']},
    'Random Forest': {'n_estimators': [100, 200], 'max_depth': [10, 20]},
    'CatBoost': {'iterations': [500, 1000], 'learning_rate': [0.01, 0.05], 'depth': [4, 6]},
}

tuned_results = {}
for name, model in models.items():
    print(f"Tuning hyperparameters for {name} with selected features...")
    grid_search = GridSearchCV(model, param_grids[name], scoring='neg_mean_squared_error', cv
    grid_search.fit(X_train_final, y_train)
    best_model = grid_search.best_estimator_

    predictions = best_model.predict(X_test_final)
    predictions_train = best_model.predict(X_train_final)
    rmse = np.sqrt(mean_squared_error(y_test, predictions))
    r2 = r2_score(y_test, predictions)
    rmse_train = np.sqrt(mean_squared_error(y_train, predictions_train))
    r2_train = r2_score(y_train, predictions_train)
    tuned_results[name] = {
        'Best Params': grid_search.best_params_,
        'RMSE Test': rmse,
        'R² Test': r2,
        'RMSE Train': rmse_train,
        'R² Train': r2_train
    }
```

Results of Hyperparameter Tuning

Model	Metric	Untuned Train	Tuned Train	Untuned Test	Tuned Test
Linear Regression	RMSE	0.1191	0.1191	0.1190	0.1190
Linear Regression	R^2	0.7736	0.7736	0.7651	0.7651
Random Forest	RMSE	0.0668	0.0396	0.1113	0.1103
Random Forest	R^2	0.9287	0.9750	0.7945	0.7981
SVR	RMSE	0.2488	0.0916	0.2441	0.1078
SVR	R^2	0.0122	0.8660	0.0124	0.8072
CatBoost	RMSE	0.0866	0.0788	0.1047	0.1044
CatBoost	R^2	0.8803	0.9009	0.8184	0.8195



Tuned Test RMSE



Tuned Test R^2

Tuned Models Evaluation

- Our Linear Regression Model performs the same, since no hyperparameters were tuned for it.
- The Random Forest Model performs slightly better on both the training and the testing data then before, but still overfits quite a bit.
- The SVR model performs much better than before, giving the second best results out of all of our models.
- Finally, the Catboost Model after tuning performs even better than the baseline and offers the best performance on the tessting data.

Conclusions

After analyzing the performance of different machine learning models for car price prediction using Mobile.de data, we can draw several key conclusions:

- Feature selection was crucial, with factors like cubic capacity, power, registration year, and vehicle make emerging as important predictors of car prices.
- The CatBoost model proved to be the most effective, achieving the best balance between performance ($R^2 = 0.8195$) and avoiding overfitting on the test data.
- While the Random Forest model showed excellent training performance ($R^2 = 0.9750$), it showed significant overfitting, making it less reliable for real-world predictions.
- Hyperparameter tuning significantly improved the SVR model's performance, transforming it from the worst-performing model to a competitive one.
- The Linear Regression model, despite its simplicity, provided consistent and reliable results ($R^2 = 0.7651$) without overfitting, making it a viable option for basic price predictions.

These results suggest that the CatBoost model would be the most suitable choice for implementing a car price prediction system on Mobile.de data, offering reliable predictions while maintaining good generalization capabilities.