Big Data Project

Group Members

- Fanica Narcis-Alexandru, Group 411
- Adam Antonio-Emanuel, Group 411
- Stefanoiu Rares, Group 407

Motivation

This project aims to develop a machine learning model for predicting car prices on Mobile.de, Europe's largest automotive marketplace. The focus is on analyzing vehicles from the Volkswagen Auto Group (VAG) manufacturers - Audi, Volkswagen, Skoda, and Seat - with prices ranging from €5,000 to €30,000.

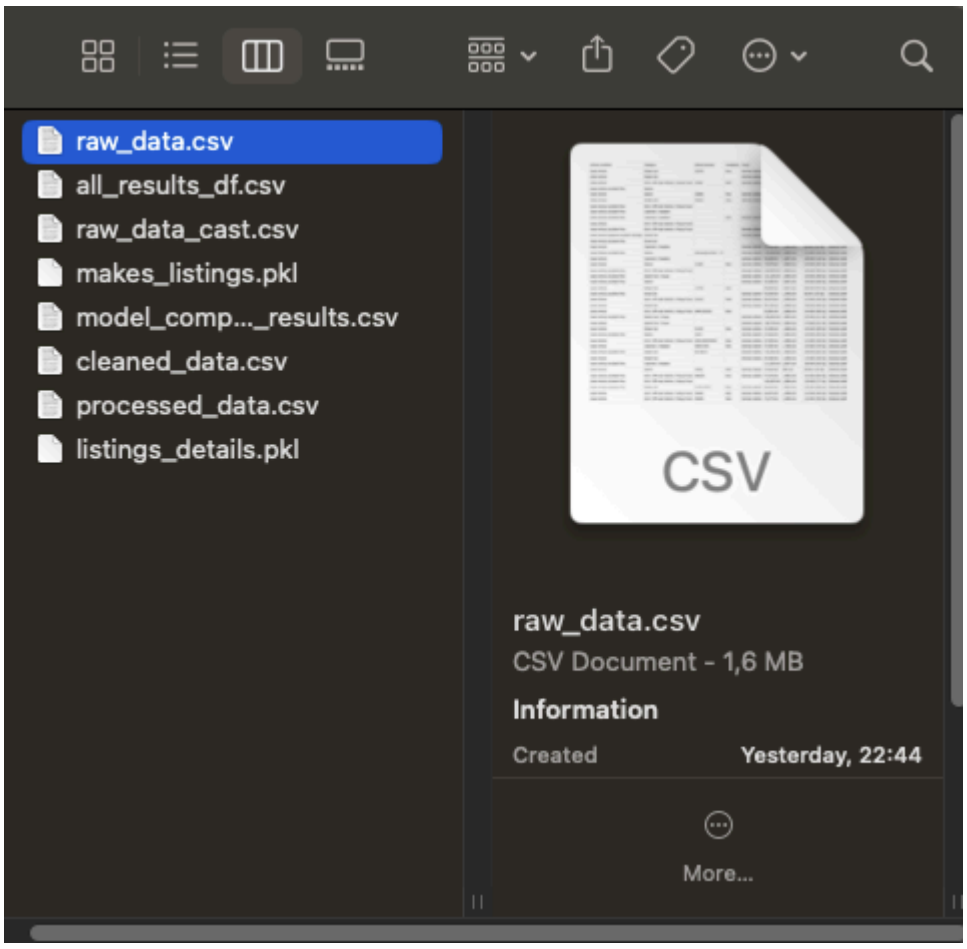
Through web scraping, data analysis, and the implementation of various regression models, we aim to identify the key factors that influence car prices and develop a reliable prediction system based on these variables.

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**:
   - At the core of the process lies the `playwright` library along with the `playwright_stealth` plugin which enable us to programatically control the browser and navigate to certain URL's as well as `BeautifulSoup` for HTML parsing and extraction of relevant data
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_singe_page` function was created to navigate a given URL and extract the HTML from that page
     - Cookie consent pop-ups were handled programmatically to ensure smooth scraping.
   - **Data Extraction**:
     - HTML content was fetched and parsed using `BeautifulSoup`.
     - More functions were created in order to extract the price and technical data of each car listing from the fetched HTML 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 listings were stored in a dictionary and organized by manufacturer.
   - Afterwards, these listings were converted to a Pandas Dataframe to allow us to perform data analysis on them.
6. **Dataset Size:**
   - The created dataset, in its raw form came out to a size of 1.6 MB




```
import os
import pickle
import random
import time
from typing import List, Dict

import pandas as pd
from tqdm import tqdm
from bs4 import BeautifulSoup
from playwright.sync_api import sync_playwright, Page, BrowserContext
from playwright_stealth import stealth_sync


def make_stealth_context(pw, headless=False) -> BrowserContext:
    """
    Launches a heavily stealthed Chromium browser.
    """
    browser = pw.chromium.launch(
        headless=headless,
        args=[
            "--disable-blink-features=AutomationControlled",
            "--start-maximized",
        ],
)
context = browser.new_context(
    user_agent=(
        "Mozilla/5.0 (Windows NT 10.0; Win64; x64) "
        "AppleWebKit/537.36 (KHTML, like Gecko) "
        "Chrome/125.0.0.0 Safari/537.36"
    ),
    viewport=None,
    locale="de-DE",
    timezone_id="Europe/Berlin",
)
stealth_sync(context)
return context
```

```
def extract_listings_from_page(soup: BeautifulSoup) → List[str]:
    try:
        listings = soup.find_all(
            "a",
            attrs={"data-testid": lambda v: v and v.startswith("result-listing-")},
        )
        if not listings:
            return []
        links = [listing["href"] for listing in listings]
        return links
    except Exception as e:
        print(f"Error extracting listings: {e}")
        return []


def scrape_single_page(page: Page, url: str) → BeautifulSoup:
    """Scrapes a single page with more human-like interactions."""
    page.goto(url, wait_until="networkidle", timeout=90000)

    cookie_button_selector = "button:has-text('Accept')"
    try:
        # Wait for the button to be potentially visible, but don't fail if not
        page.wait_for_selector(cookie_button_selector, timeout=5000)
        cookie_button = page.locator(cookie_button_selector).first
        if cookie_button.is_visible():
            # Move mouse over the button first
            cookie_button.hover()
            time.sleep(random.uniform(0.2, 0.3))
            cookie_button.click()
            # Wait for the overlay to disappear
            time.sleep(random.uniform(0.3, 0.5))
    except Exception:
        print("Cookie button not found or timed out, continuing.")
        pass

    # Simulate reading time
    time.sleep(random.uniform(0.2, 0.5))

    html = page.content()
    return BeautifulSoup(html, "html.parser")


def scrape_all_make_pages(
    context: BrowserContext, make_id: str, max_pages: int = 50
) → List[str]:
    """Scrapes all pages for a given make ID with slower, safer pacing."""
    base_url = f"https://suchen.mobile.de/fahrzeuge/search.html?dam=false&isSearchRequest=true&s=Car&sb=rel&vc=Ca
    all_listings = []
    page = context.new_page()

    for page_num in tqdm(range(1, max_pages + 1), desc=f"Scraping {make_id}"):
        url = f"{base_url}&pageNumber={page_num}"
        soup = scrape_single_page(page, url)
        relative_links = extract_listings_from_page(soup)

        if relative_links:
            absolute_links = [
```

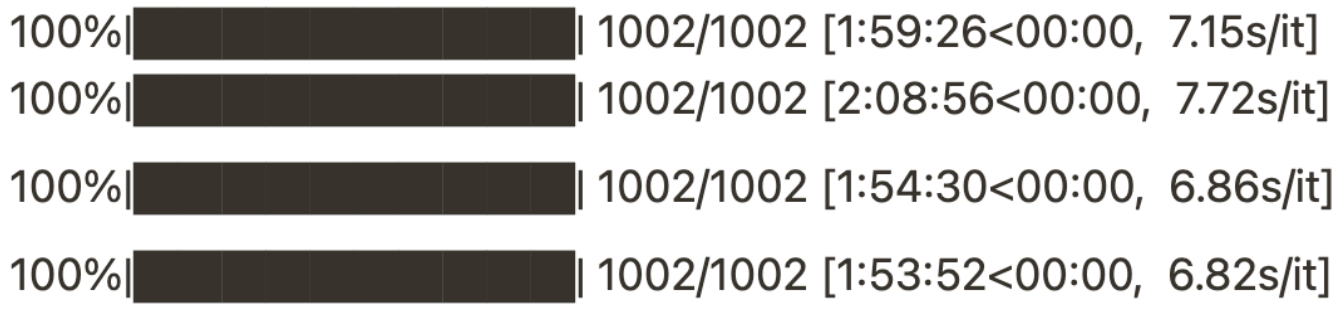
```
            f"https://suchen.mobile.de{link}" for link in relative_links
        ]
        all_listings.extend(absolute_links)
    else:
        print(f"No more listings found on page {page_num}. Stopping.")
        break

    print("Taking a longer break between search pages...")
    time.sleep(random.uniform(8.0, 15.0))

page.close()
return all_listings


def extract_price_from_listing(soup: BeautifulSoup):
    try:
        car_price_div = soup.find("div", attrs={"data-testid": "vip-price-label"})
        car_price = car_price_div.find("div").text
        return car_price
    except (AttributeError, IndexError):
        return "Price not found"


def extract_technical_details_from_listing(soup: BeautifulSoup):
    try:
        car_data_article = soup.find(
            "article", attrs={"data-testid": "vip-technical-data-box"}
        )
        car_data_dl = car_data_article.find("dl")
        car_data_items = car_data_dl.find_all(["dt", "dd"])
        car_data_pairs = list(zip(car_data_items[::2], car_data_items[1::2]))
        return {dt.text.strip(): dd.text.strip() for dt, dd in car_data_pairs}
    except (AttributeError, IndexError):
        return {}


def scrape_all_listings_for_make(
    pw, make_listings: List[str], make_name: str
) -> List[Dict]:
    all_details = []
    for listing_url in tqdm(make_listings, desc=f"Scraping details for {make_name}"):
        try:  # Create a new browser context for each listing
            context = make_stealth_context(pw, headless=False)
            page = context.new_page()

            soup = scrape_single_page(page, listing_url)
            if not soup.find("div", attrs={"data-testid": "vip-price-label"}):
                print(f"Skipping invalid or blocked listing: {listing_url}")
                time.sleep(random.uniform(15.0, 25.0))
                context.close()
                context.browser.close()
                continue

            details = {
                "make": make_name,
                "price": extract_price_from_listing(soup),
                "technical_details": extract_technical_details_from_listing(soup),
            }
```

```
            all_details.append(details)

            page.close()
            context.close()
            context.browser.close()

            time.sleep(random.uniform(1.0, 2.0))
        except Exception as e:
            print(f"Error scraping listing {listing_url}: {e}")
            time.sleep(random.uniform(1.0, 2.0))
            try:
                page.close()
                context.close()
                context.browser.close()
            except:
                pass
            continue
    return all_details


def load_and_clean_data() -> pd.DataFrame:
    with open("./data/listings_details.pkl", "rb") as f:
        data = pd.read_pickle(f)
    for make, records in data.items():
        for record in records:
            record["Make"] = make
    raw_data = []
    for make, records in data.items():
        raw_data.extend(records)
    raw_data = pd.DataFrame(raw_data)
    raw_data.to_csv("./data/raw_data.csv", index=False)
    return raw_data


if __name__ == "__main__":
    makes = {
        "Audi": "1900",
        "Volkswagen": "25200",
        "Skoda": "22900",
        "Seat": "22500",
    }
    with sync_playwright() as pw:
        # --- Part 1: Scrape listing URLs ---
        all_make_links = {}
        all_make_links: Dict[str, List[str]] = {}
        if os.path.exists("./data/makes_listings.pkl"):
            with open("./data/makes_listings.pkl", "rb") as f:
                all_make_links = pickle.load(f)
        else:
            context = make_stealth_context(pw, headless=False)
            for make, make_id in makes.items():
                make_links = scrape_all_make_pages(context, make_id, max_pages=50)
                all_make_links[make] = make_links
            with open("./data/makes_listings.pkl", "wb") as f:
                pickle.dump(all_make_links, f)
            context.close()
            context.browser.close()
```

```
    # --- Part 2: Scrape details ---
    all_listings_details = {}
    for make, links in all_make_links.items():
        if links:
            listings_details = scrape_all_listings_for_make(pw, links, make)
            all_listings_details[make] = listings_details
    with open("./data/listings_details.pkl", "wb") as f:
        pickle.dump(all_listings_details, f)

    load_and_clean_data()
```

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



Webscraping Process Progress

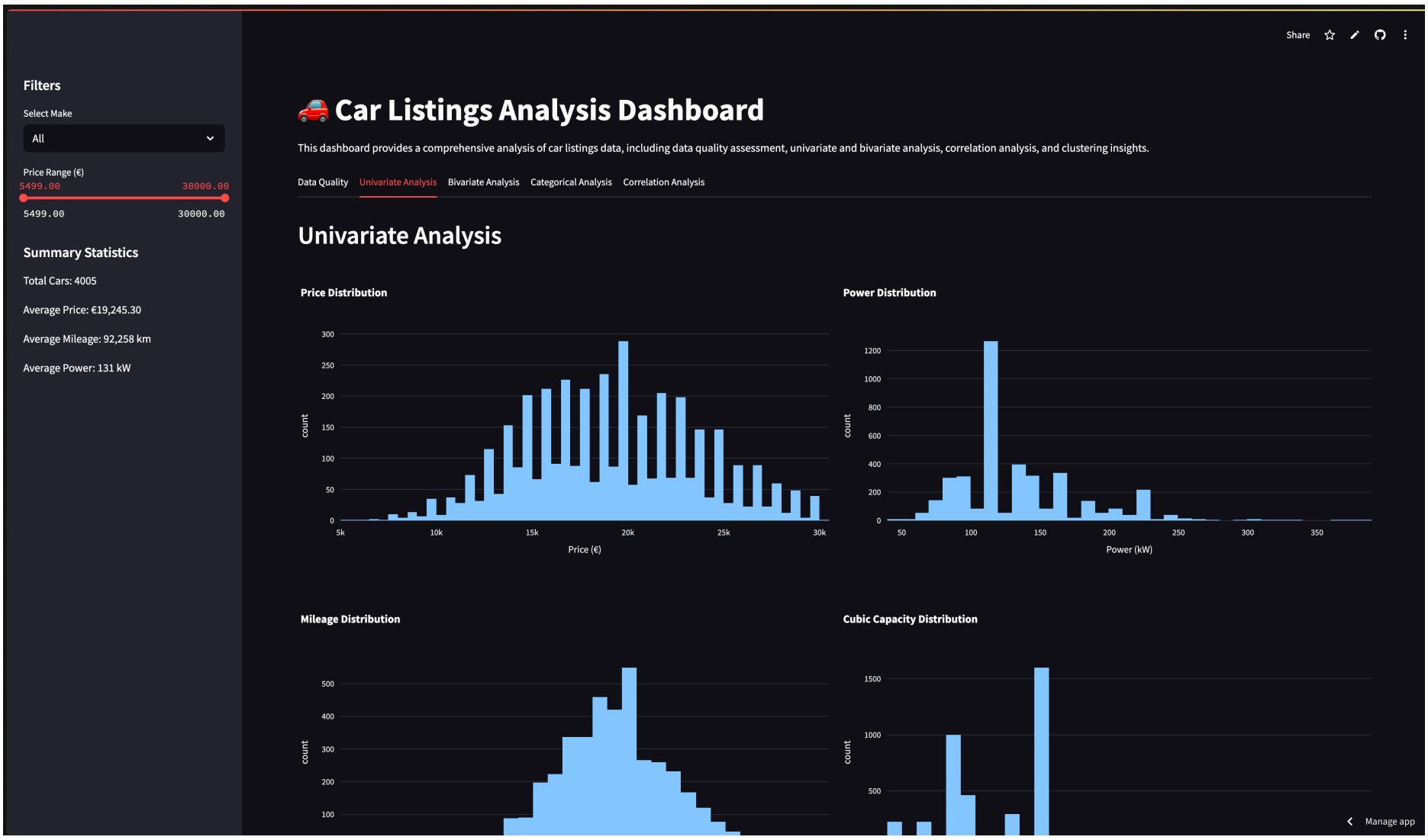
Exploratory Data Analysis

Dashboard

To better visualize the data analysis project, a streamlit dashboard was created to include the most relevant plots from the analysis process.

It is deployed on the Steamlit Community Cloud and it can be accessed through the following URL:

https://big-data-car-listings.streamlit.app/



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).
- **First Registration:** Month and year when the car was made
- **Category:** Saloon/SUV/Estate/Hatchback etc.
- 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

Column Name	Non-null Count	Type
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

Conclusions after our first look at the dataset

- We have just over 4000 rows in out dataset, corresponding to 4000 listings, around 1000 for each of the 4 makes for which we performed the webscraping process
- We have just shy of 50 features. However these are all treated as strings ('object' type) since that is how they were extracted from the scraped HTML.
- We should also take note to the fact that many of these features have missing records, so we will need to analyze this in more detail.

Data Cleaning For Numeric Features

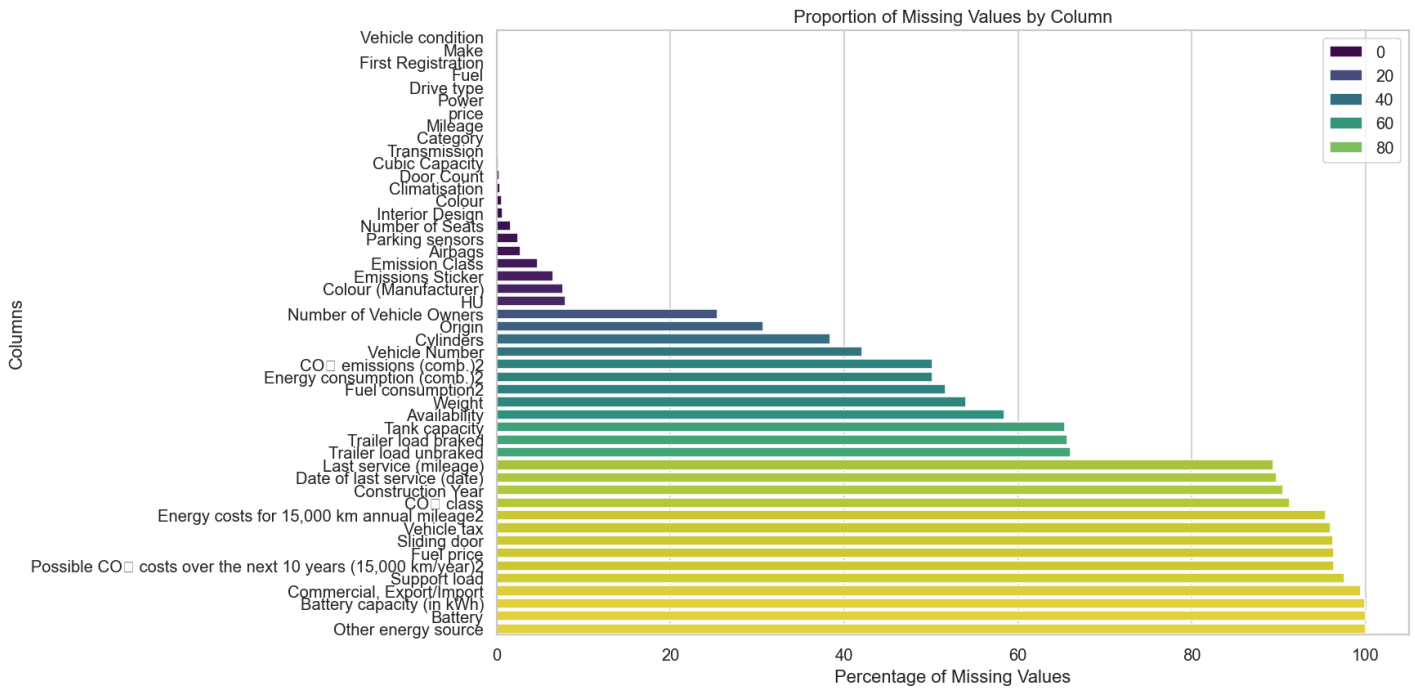
```
# Clean Numeric Columns
def clean_numeric_column(column, remove_text=True):
    if remove_text:
```

```
        return column.str.replace(r"[^\d.]", "", regex=True).astype(float)
    return column
```

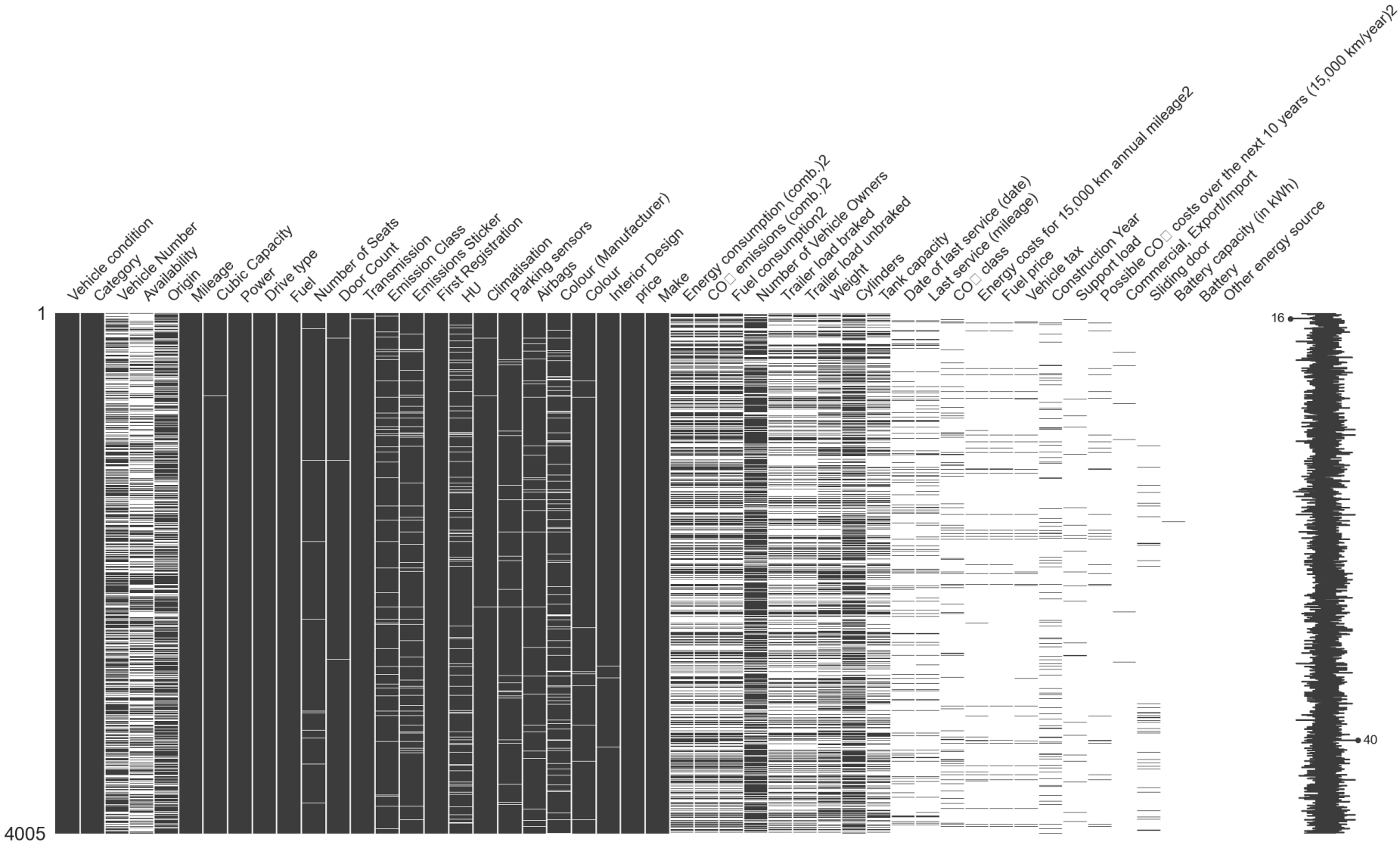
```
# Convert  columns to numeric
raw_data["price"] = clean_numeric_column(raw_data["price"])
raw_data["Mileage"] = clean_numeric_column(raw_data["Mileage"])
raw_data["Cubic Capacity"] = clean_numeric_column(raw_data["Cubic Capacity"])
raw_data["Power"] = raw_data["Power"].str.extract(r"(\d+)", expand=False).astype(float)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

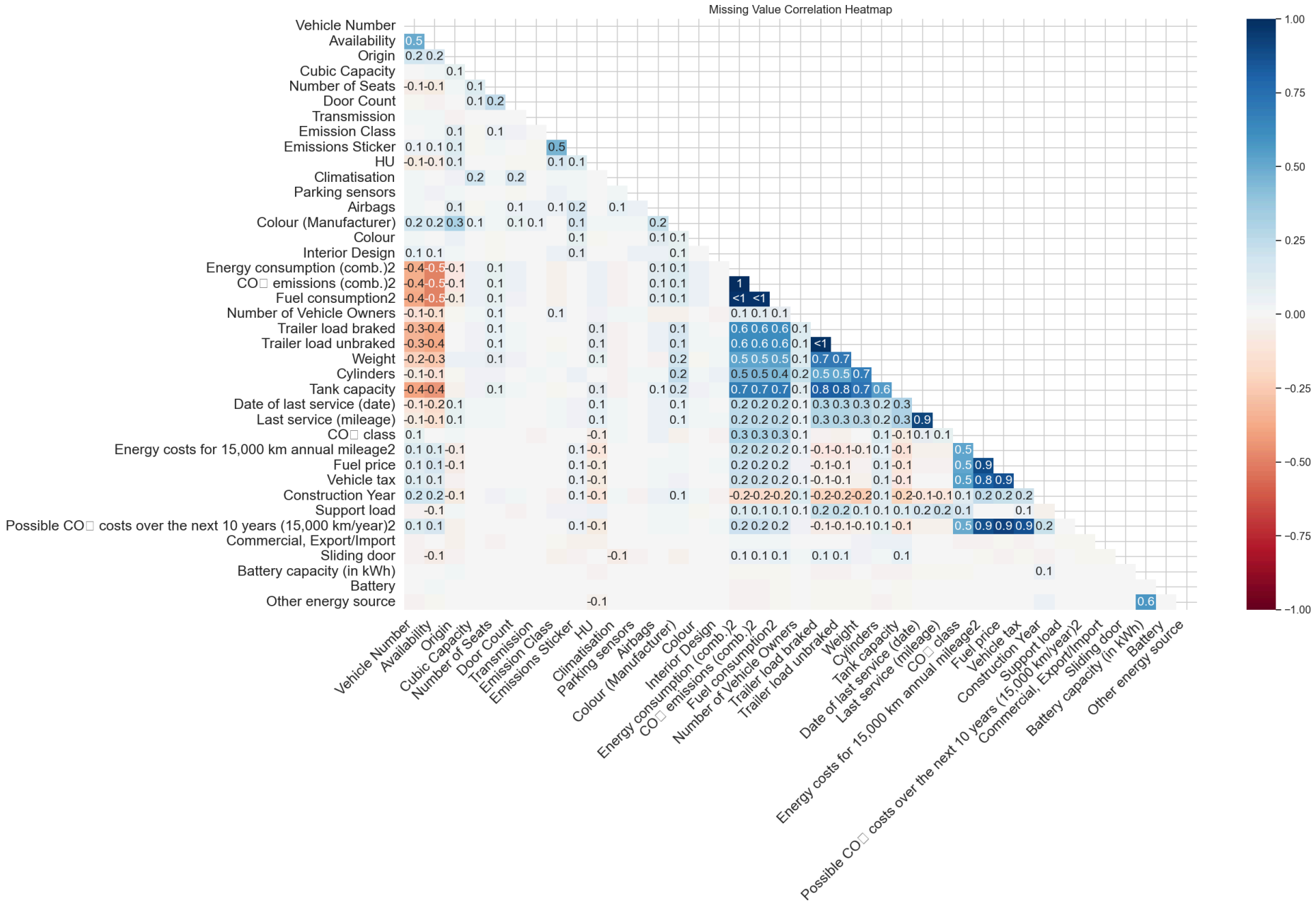
Missing Records Analysis

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.



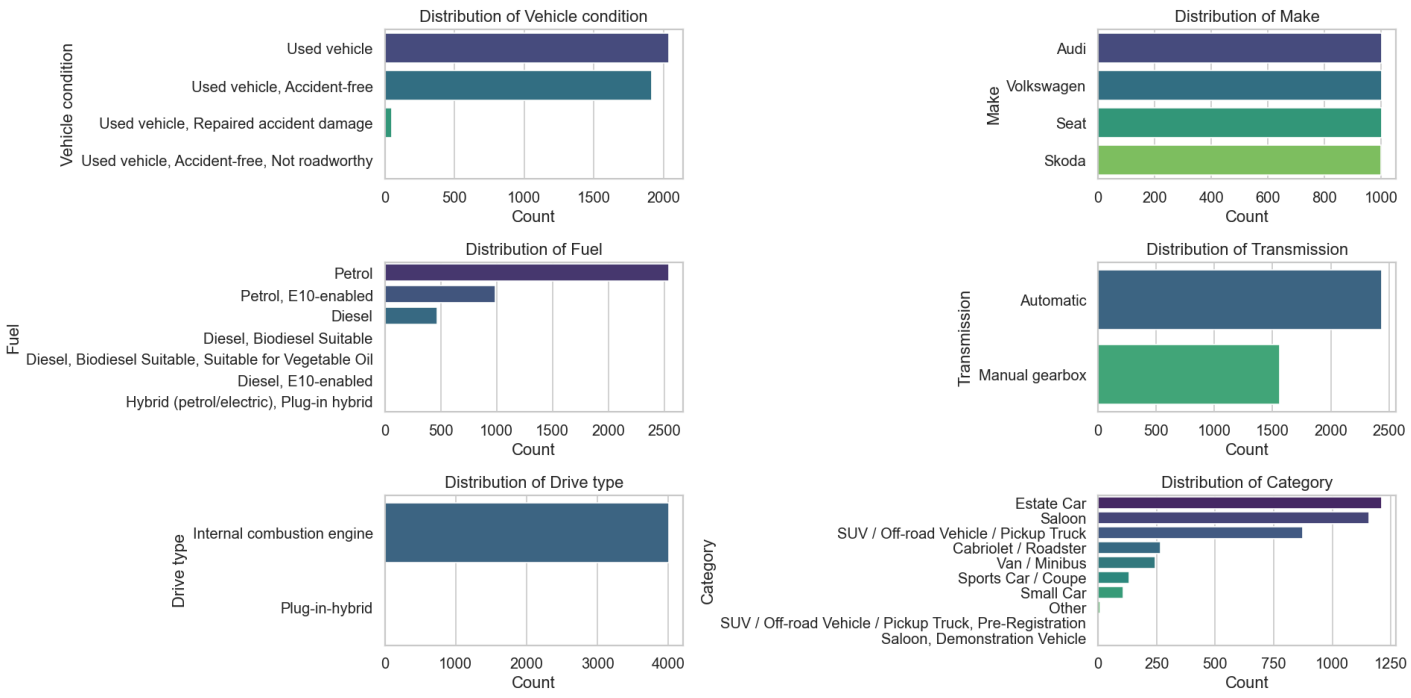
Missing Value Matrix





Categorical Features

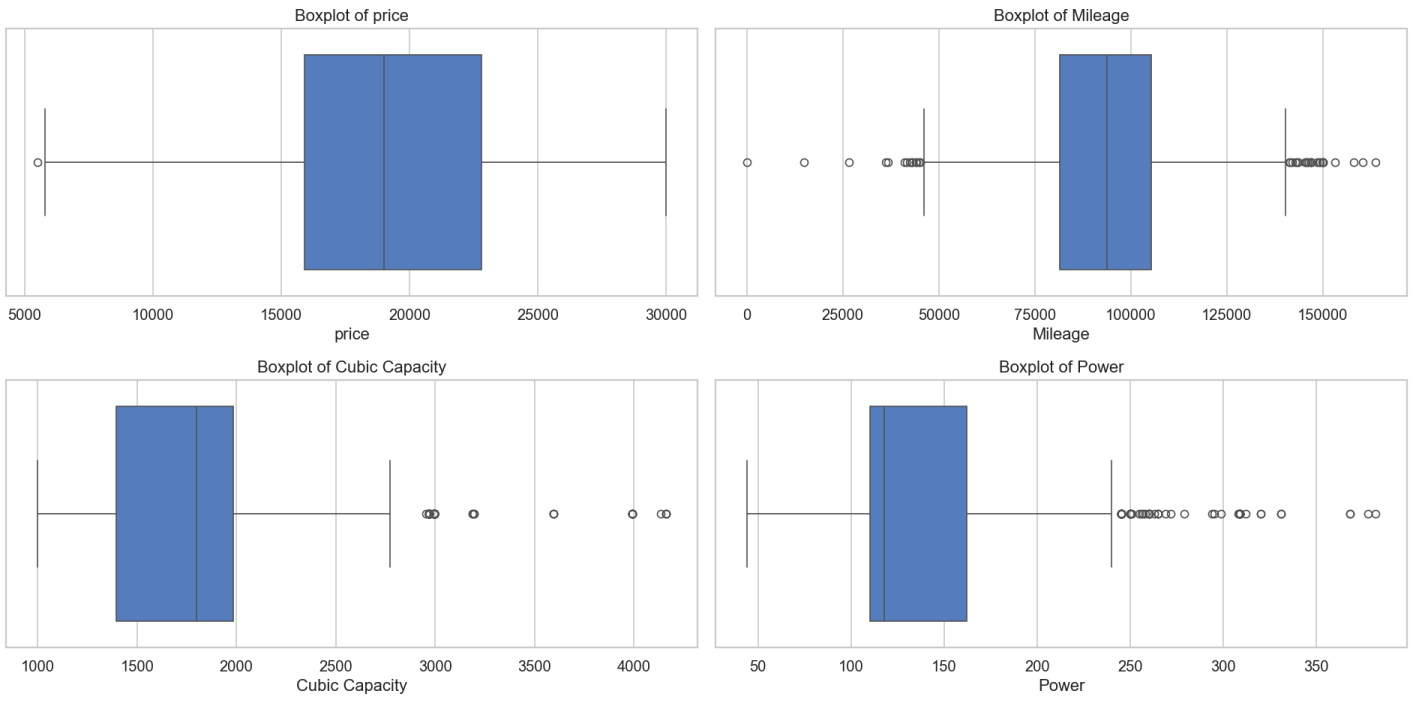
With the numeric features out of the way, we can take a quick look at the categorical ones, more specifically, their distribution

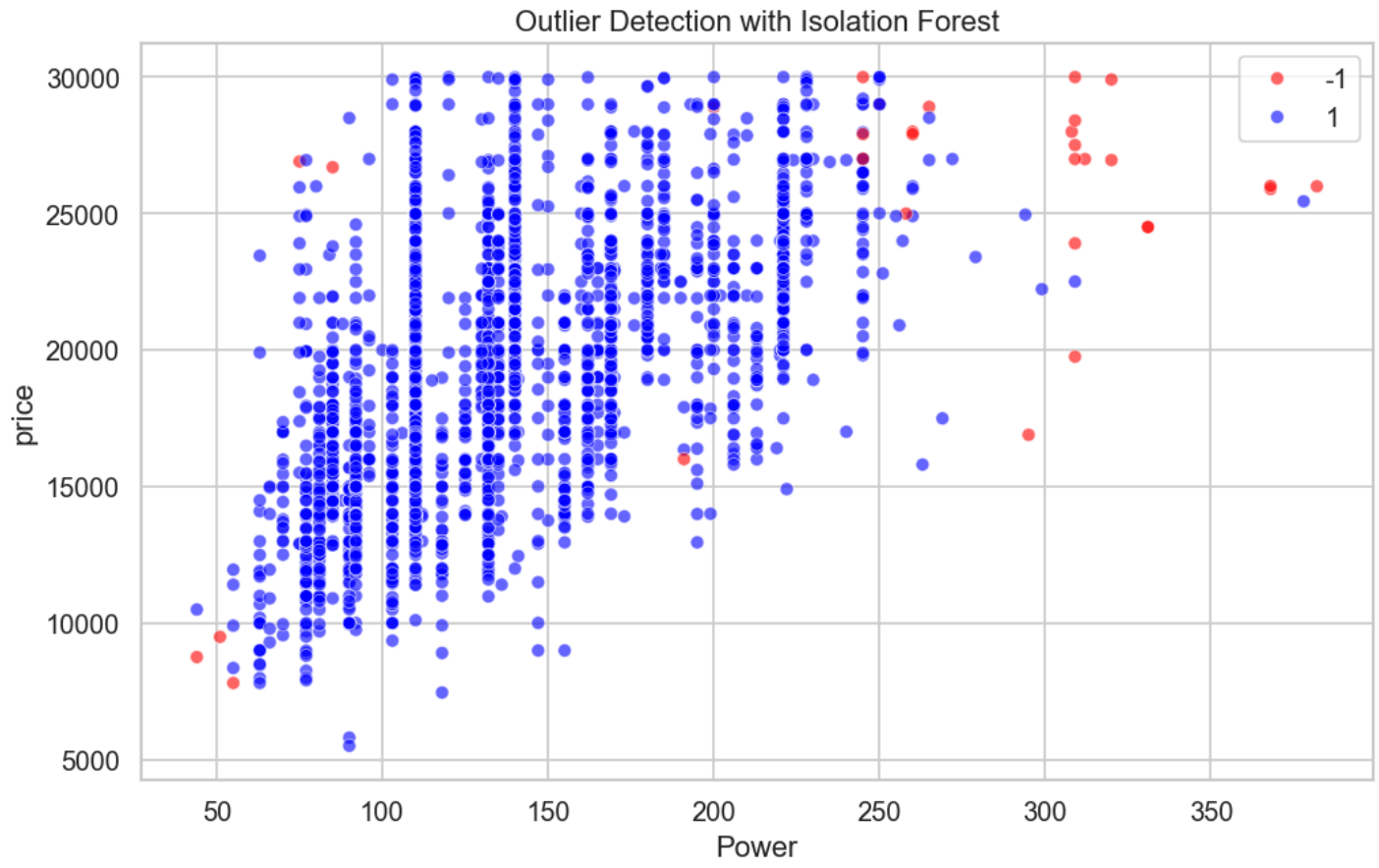


- We notice the makes are well distributed across the dataset, which is to be expected.
- Another thing to be expected, since the price range caps of at 30000, is that almost all car listings only have an internal combustion engine, with a few having a plug-in-hybrid system.
- Since Diesel engines are 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 as well as Isolation Forest



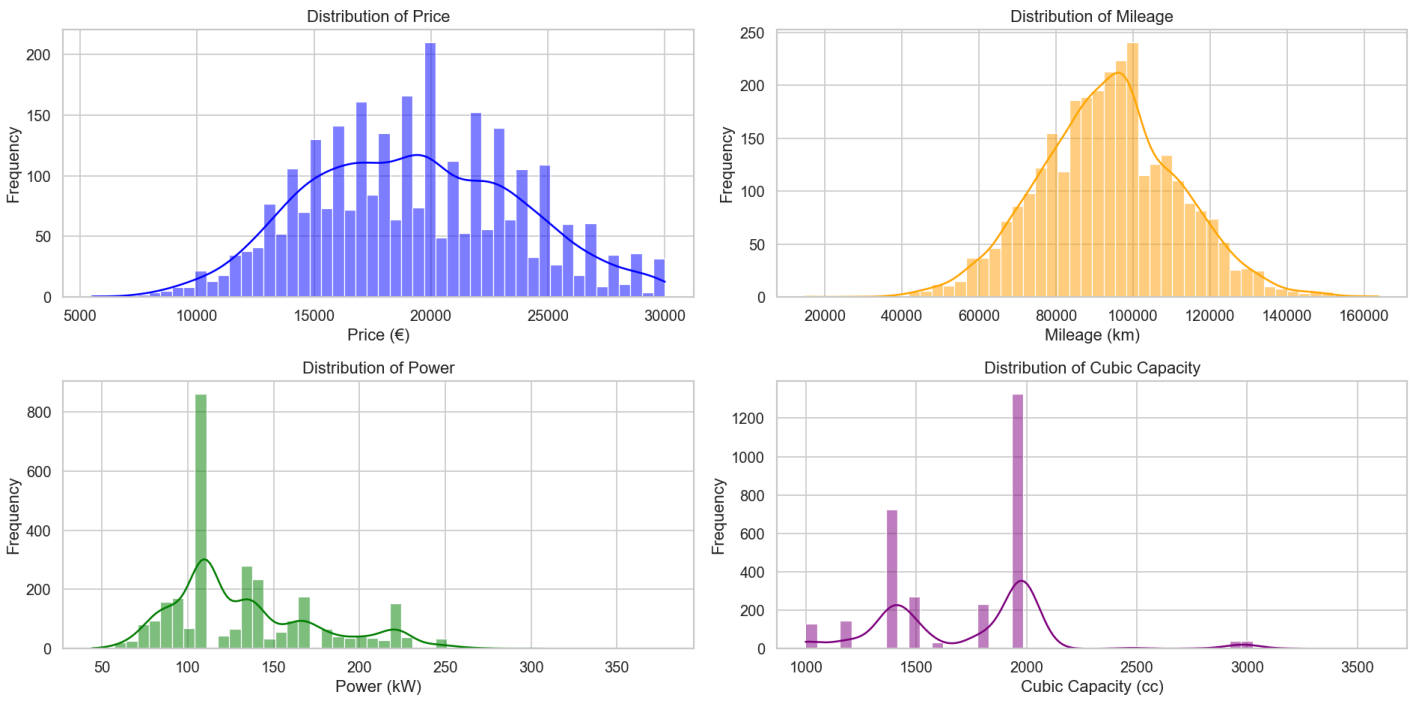


To handle these outliers, we use Isolation Forest

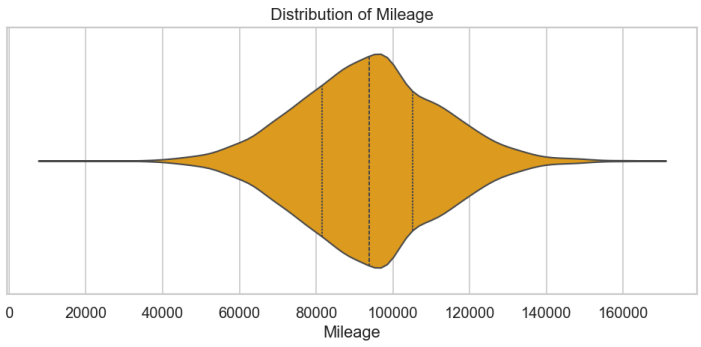
```
# Detect Outliers
iso = IsolationForest(contamination=0.01, random_state=42)
outliers = iso.fit_predict(raw_data_cleaned[numeric_columns])
# Drop outliers
raw_data_cleaned = raw_data_cleaned[outliers == 1]
```

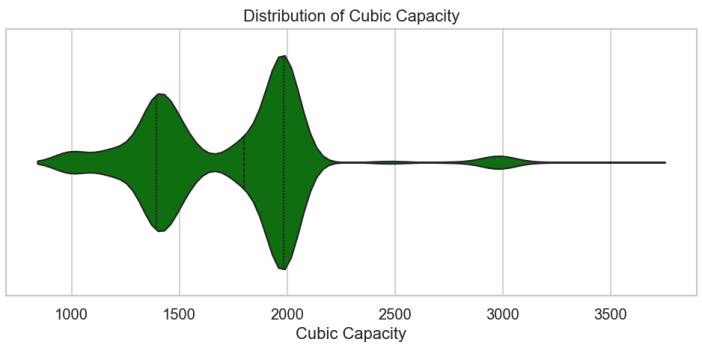
Univariate Analysis

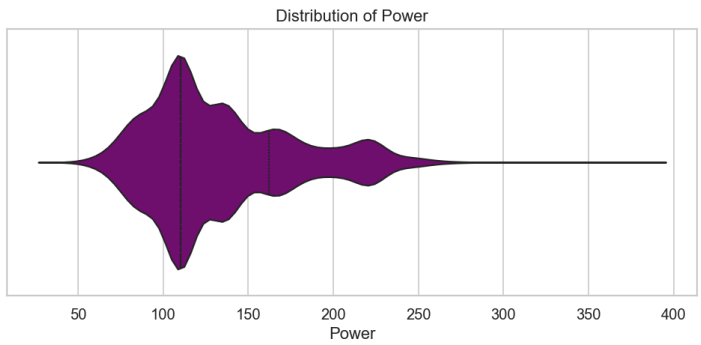
- We can visualize the distribution of our key numerical and categorical features to better understand them



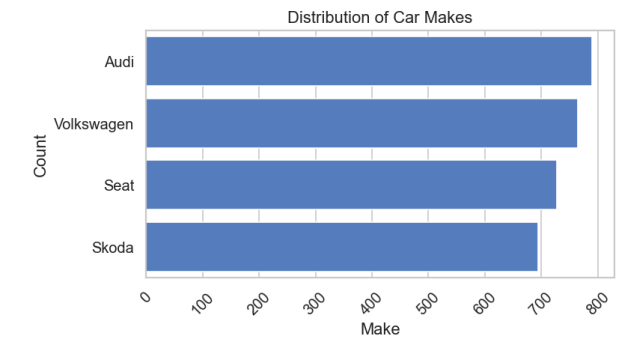
Distribution of Key Numeric Features

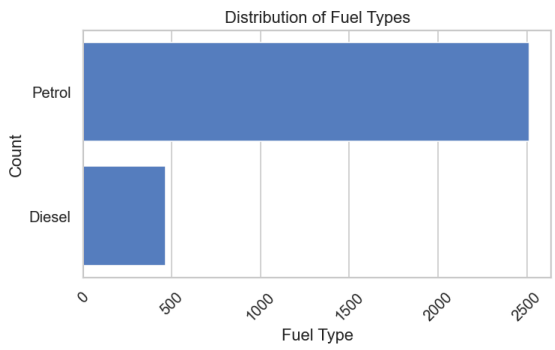


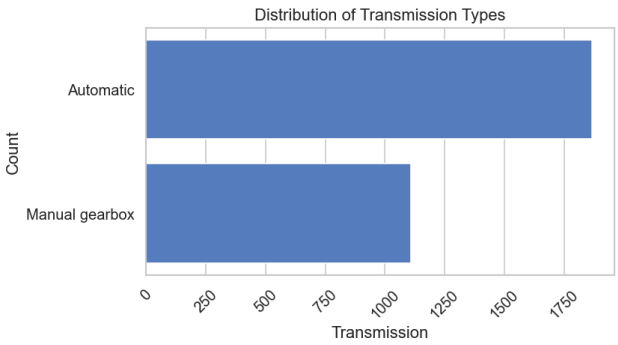


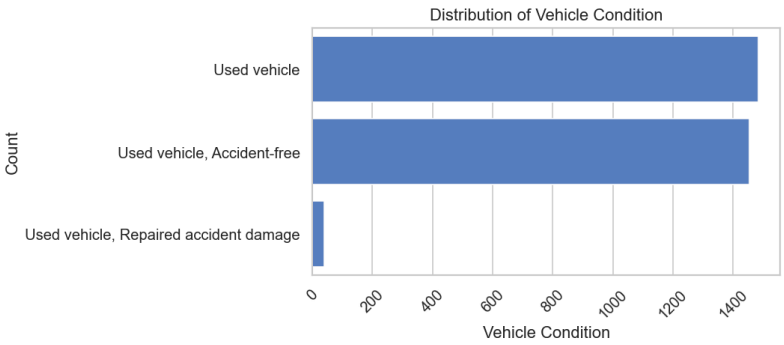


Distribution of Key Numeric Features

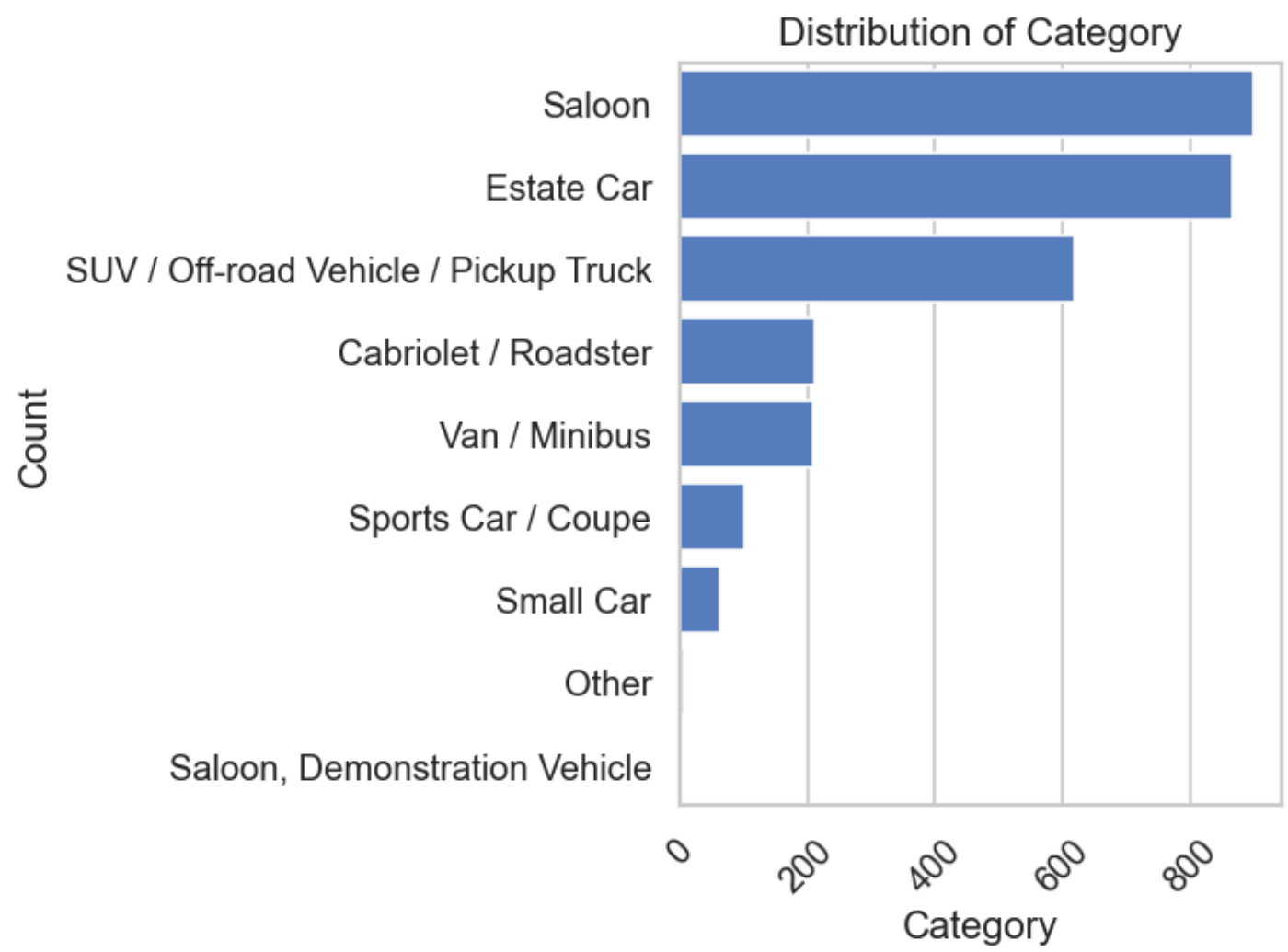








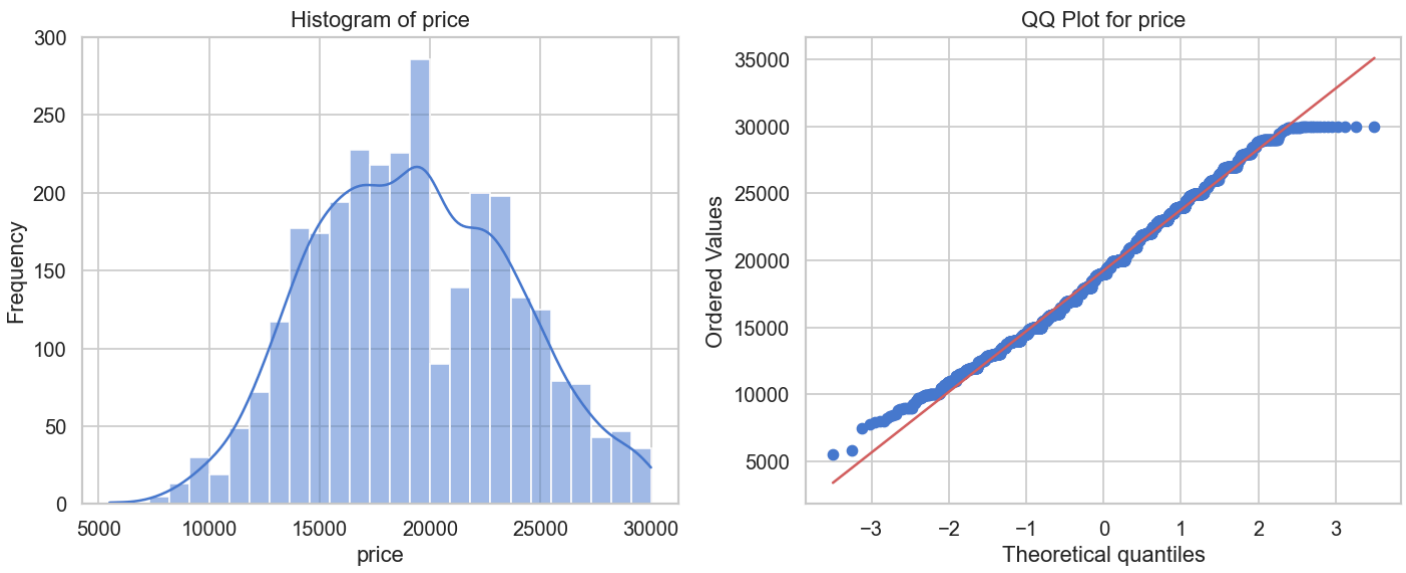
Distribution of Key Categorical Features



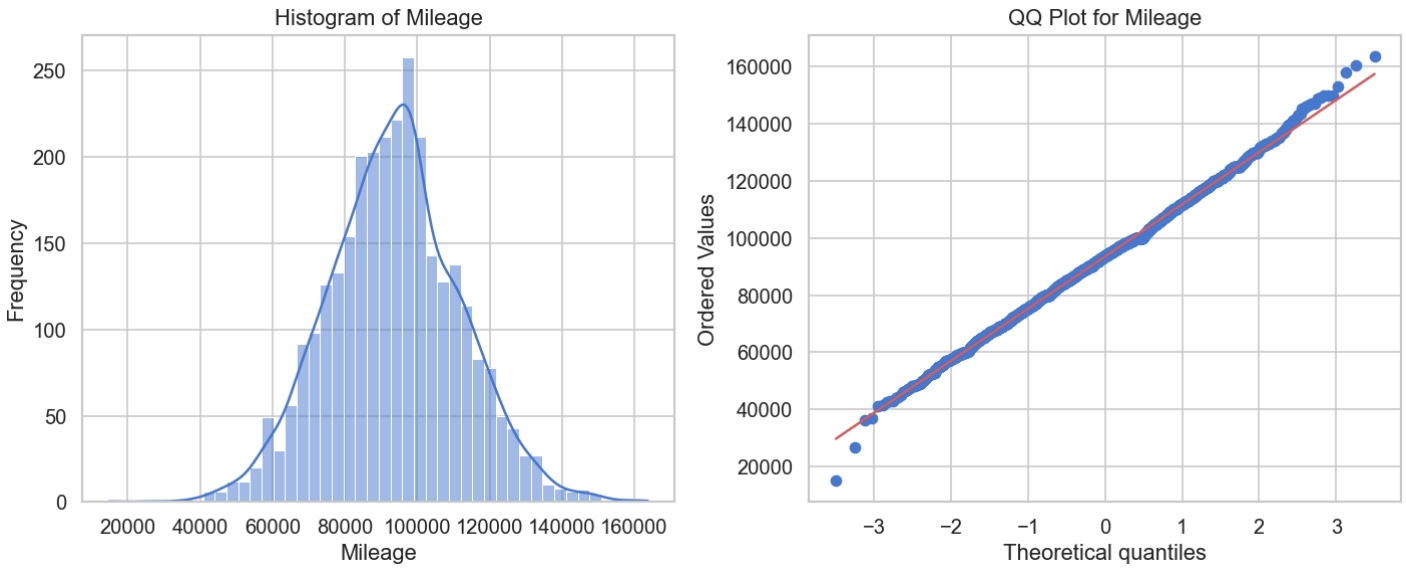
Distribution of Key Categorical Features

We can also create QQ plots for the numeric features as such:

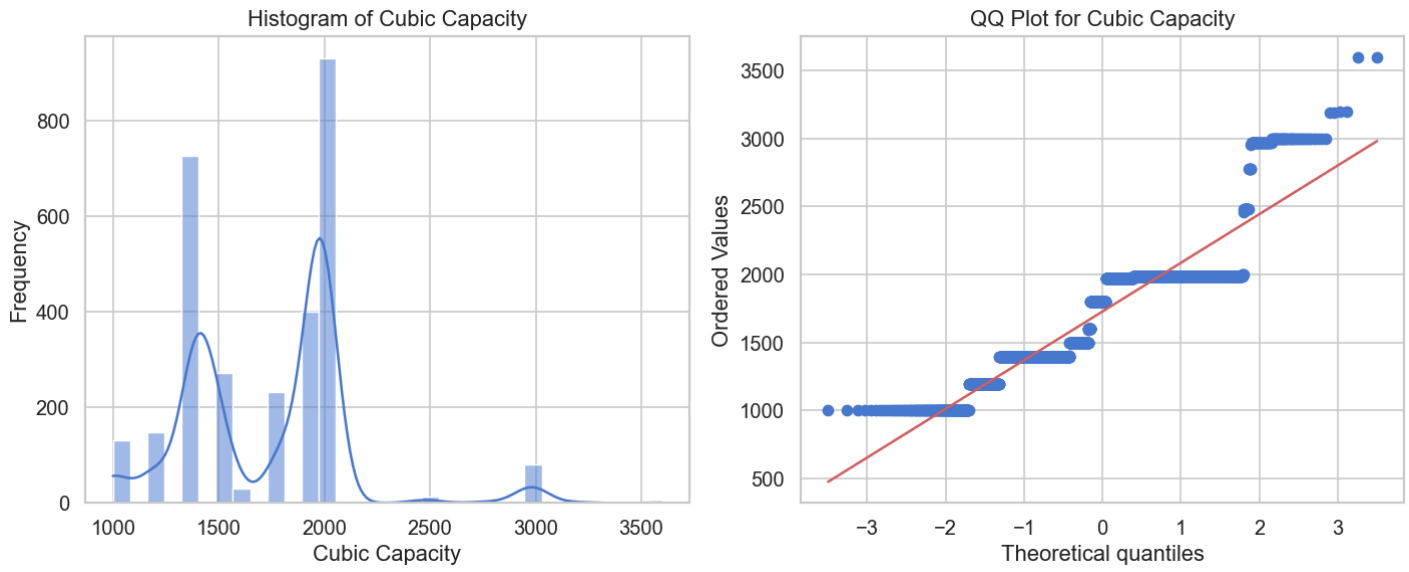
```
plt.figure(figsize=(16, 8))
for col in numeric_columns:
    print(
        f"{col}: Skewness={skew(raw_data_cleaned[col]):.2f}, Kurtosis={kurtosis(raw_data_cleaned[col]):.2f}"
    )
    stat, p = shapiro(raw_data_cleaned[col].dropna())
    print(f"Shapiro-Wilk test for {col}: p-value={p:.4f}")

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    sns.histplot(raw_data_cleaned[col].dropna(), kde=True, ax=axes[0])
    axes[0].set_title(f"Histogram of {col}")
    axes[0].set_xlabel(col)
    axes[0].set_ylabel("Frequency")

    probplot(raw_data_cleaned[col].dropna(), dist="norm", plot=axes[1])
    axes[1].set_title(f"QQ Plot for {col}")
    plt.tight_layout()
    plt.show()
```



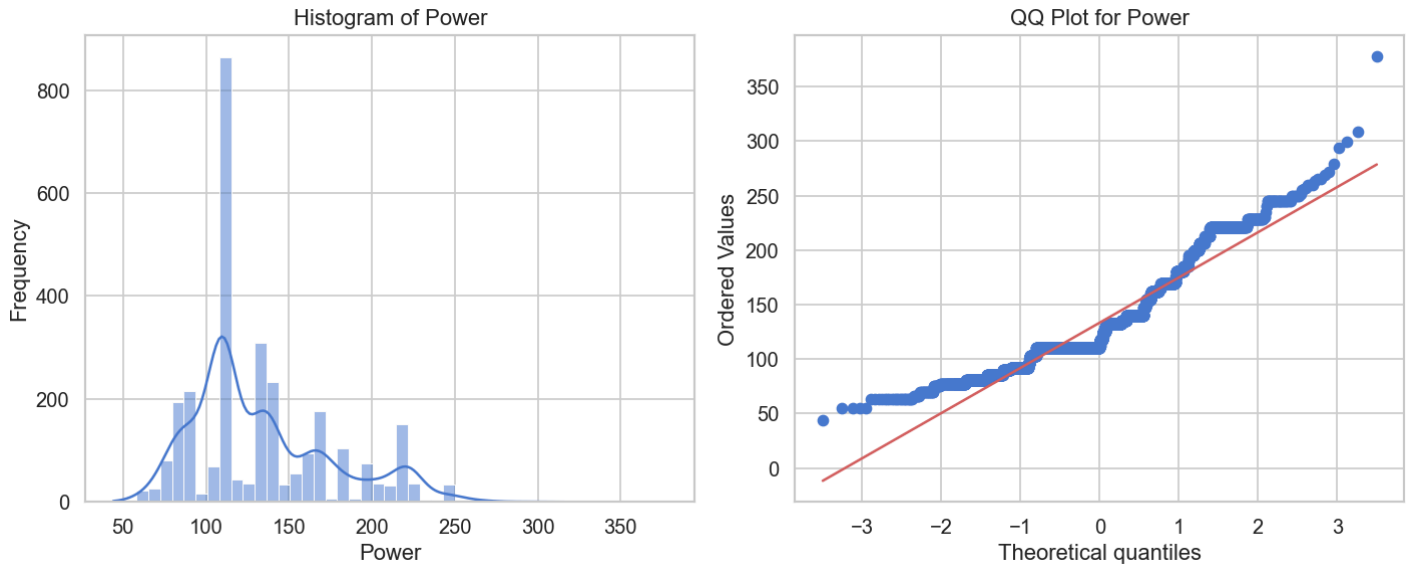
Histogram and QQ plot of Price



Histogram and QQ plot of Mileage



Histogram and QQ plot of Cubic Capacity



Histogram and QQ plot of Power (Kw)

Univariate Analysis Conclusions

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.
5. Make
   - The number for all 4 makes is roughly the same, which is to be expected since we scraped 1000 listings for each make.
   - Some of the records have been removed due to outliers.
6. Fuel Type
   - There are far more Petrol fueled cars than Diesel fueled ones, again, to be expected since most modern cars come with Petrol engines.

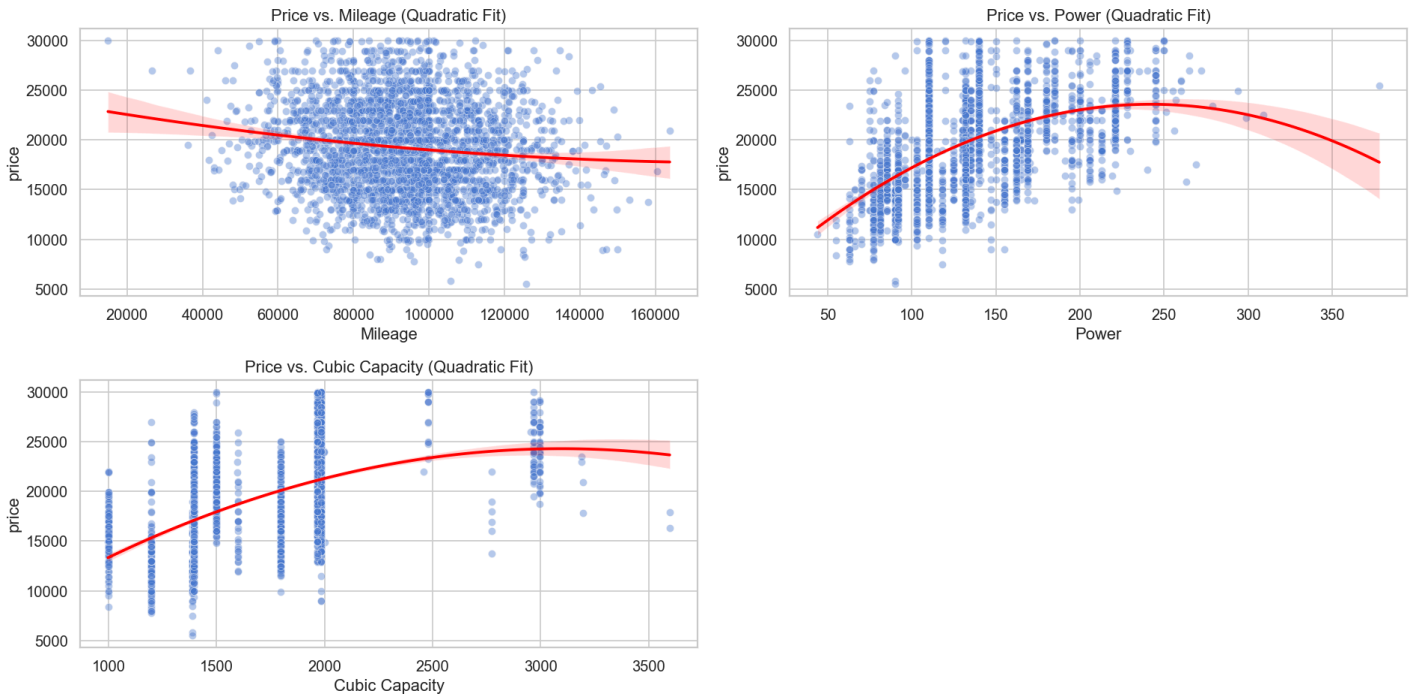
7. Transmission Type
    - More automatic transmission than manual ones, which may also lead to higher prices for those cars.
8. Category
    - Usual categories are the most common ones, including Saloon and Estate with Sports cars and Small hatchbacks being less popular.

Bivariate Analysis

Purpose:

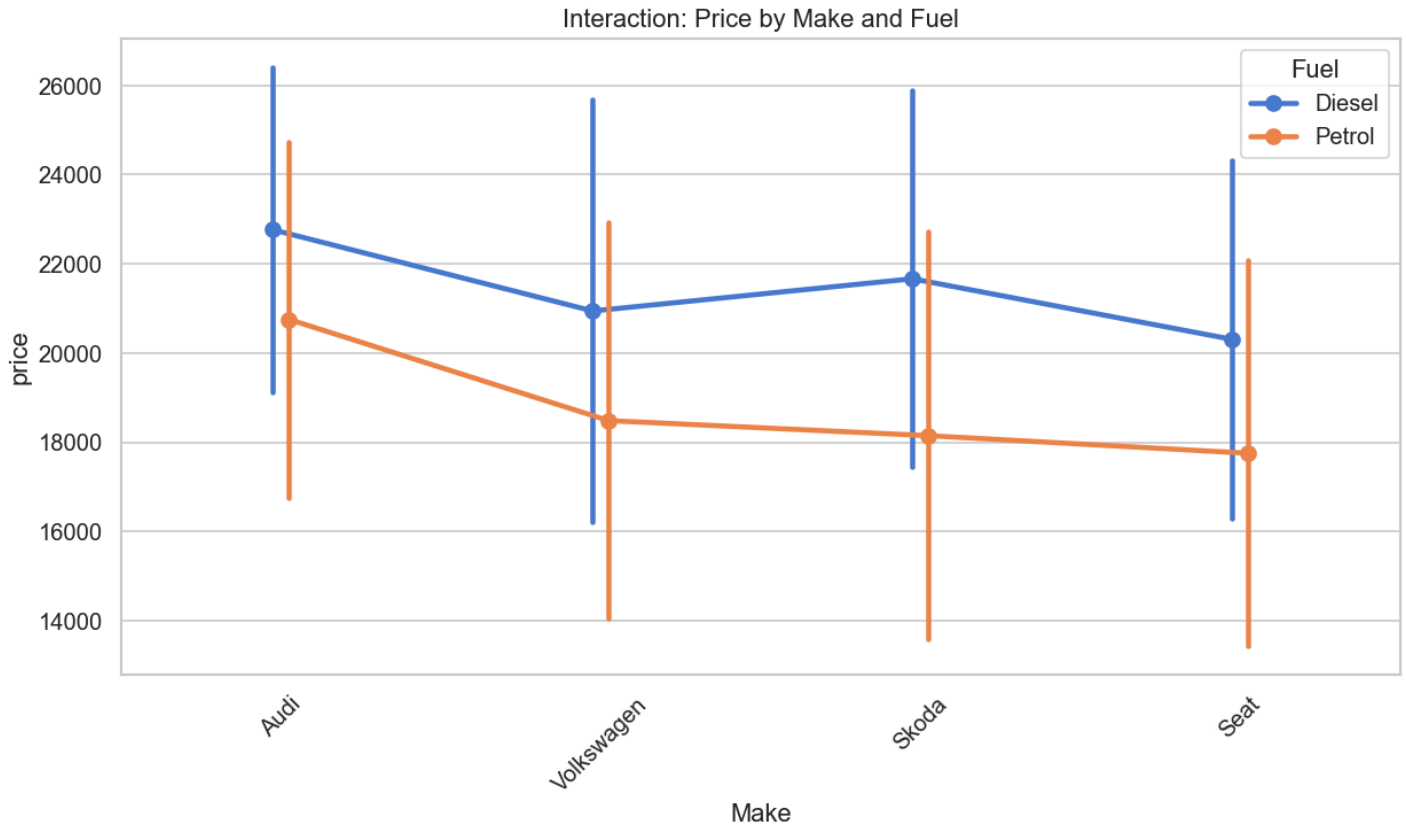
To explore relationships between pairs of variables, such as:

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

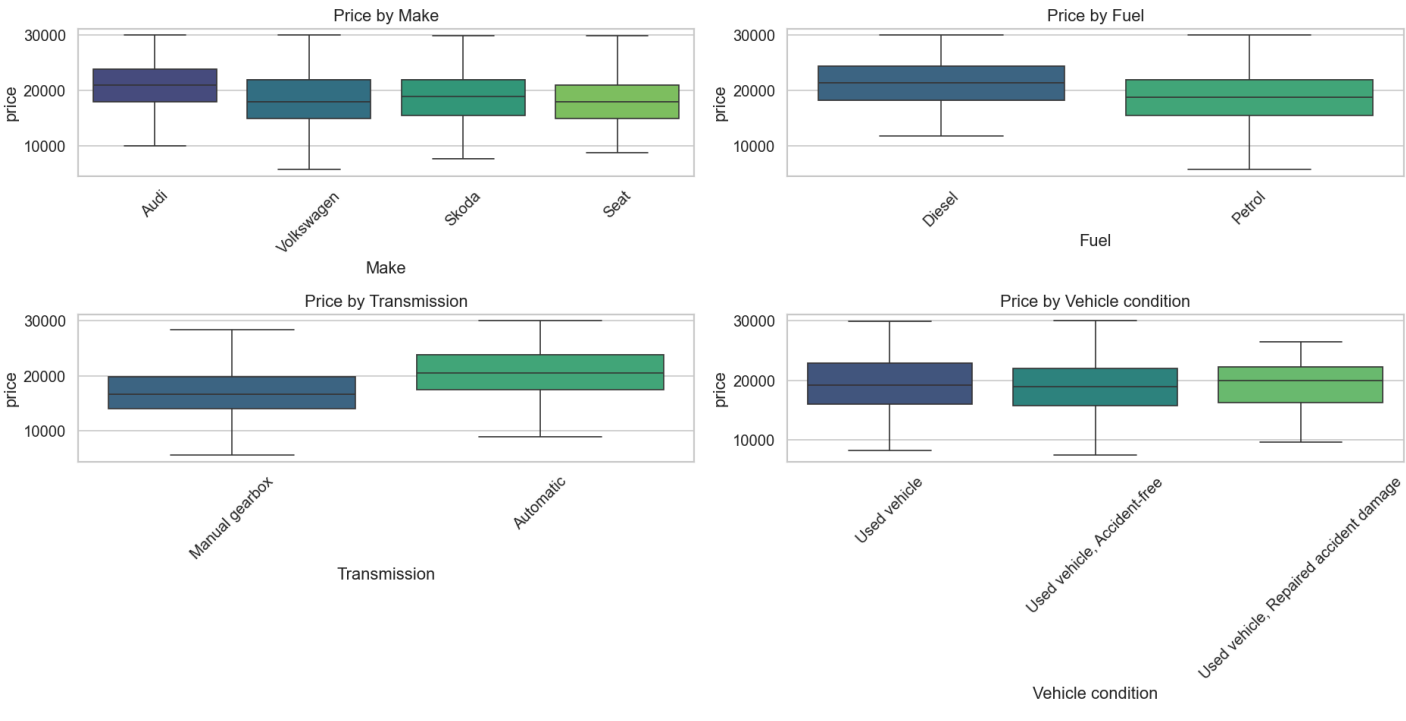


Analysis of Price and numeric features

```
# Interaction Plot: Price by Make and Fuel
plt.figure(figsize=(12, 6))
sns.pointplot(
    data=raw_data_cleaned, x="Make", y="price", hue="Fuel", dodge=True, errorbar="sd"
)
plt.title("Interaction: Price by Make and Fuel")
plt.xticks(rotation=45)
plt.show()
```



Interaction of make and fuel



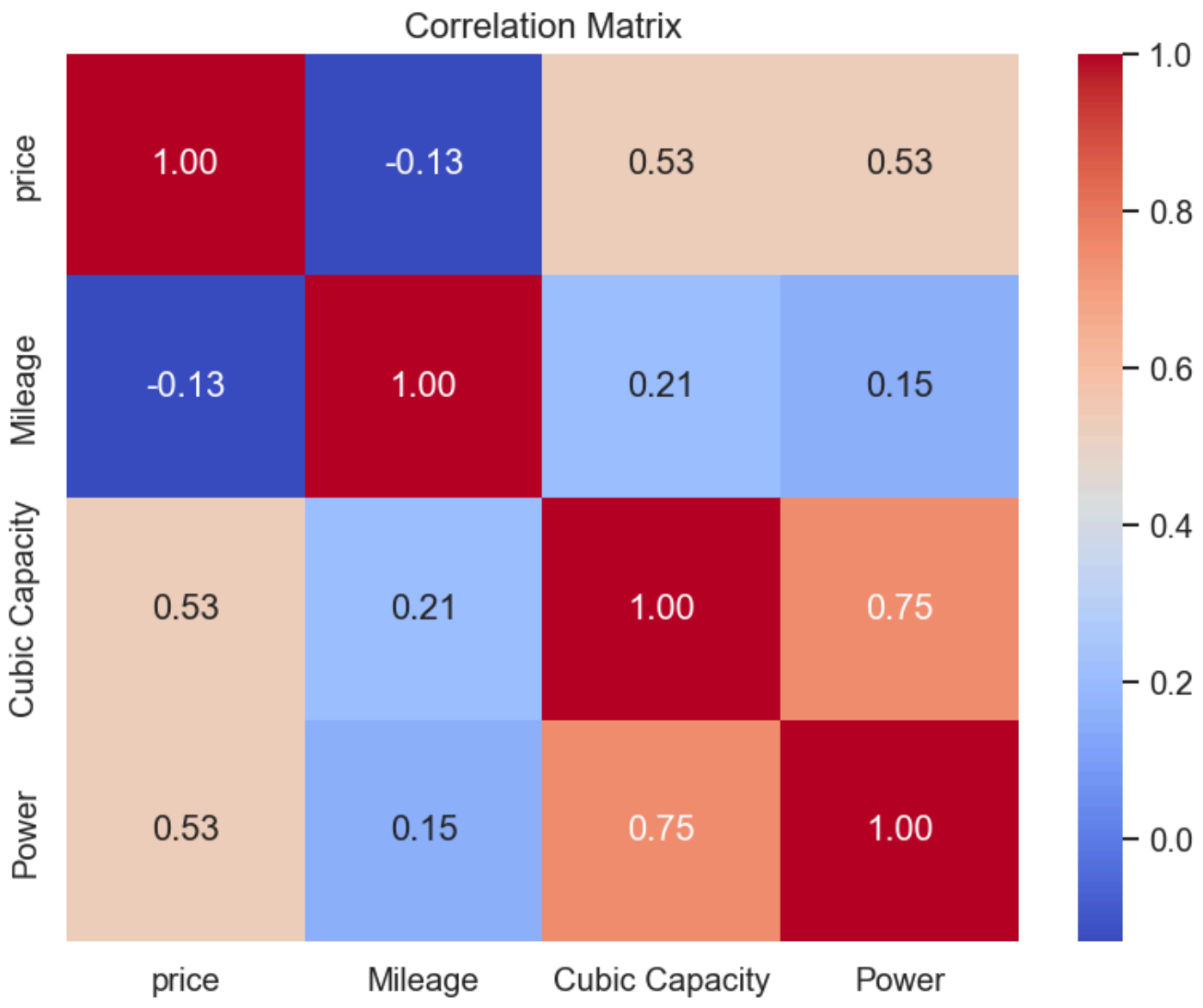
Analysis of Price and categorical features

Correlation Analysis

Purpose:

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

```
numeric_cols = ["price", "Mileage", "Cubic Capacity", "Power"]
correlation_matrix = raw_data_cleaned[numeric_cols].corr()

plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation Matrix")
plt.show()
```



Bivariate Analysis Conclusions

1. 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).

1. 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.

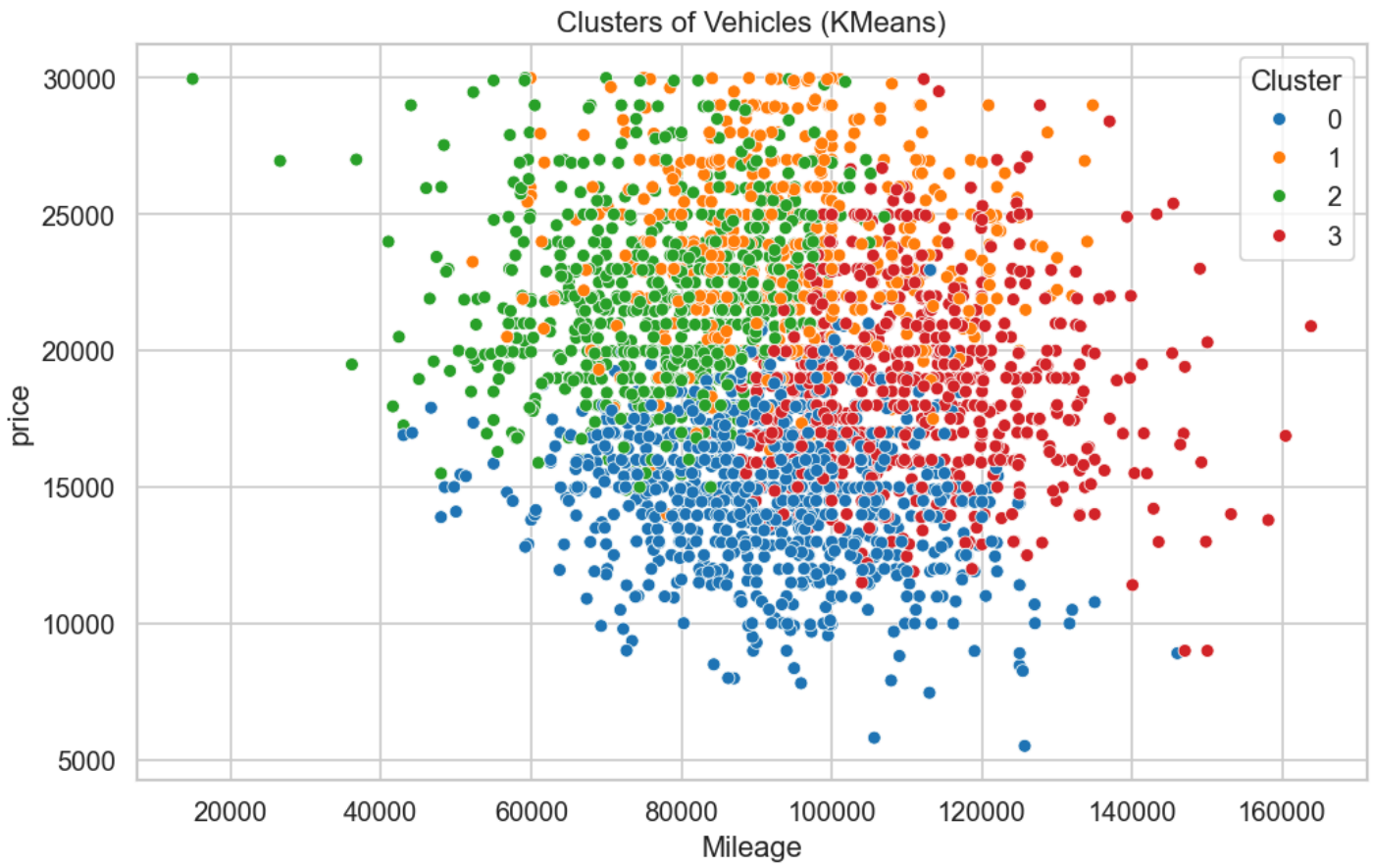
1. Correlation Analysis

- 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.

Cluster Analysis

```
# Clustering for Unsupervised Exploration
scaler = StandardScaler()
X_scaled = scaler.fit_transform(raw_data_cleaned[numeric_columns])
kmeans = KMeans(n_clusters=4, random_state=42)
clusters = kmeans.fit_predict(X_scaled)
raw_data_cleaned["Cluster"] = clusters

plt.figure(figsize=(10, 6))
sns.scatterplot(
    x=raw_data_cleaned["Mileage"],
    y=raw_data_cleaned["price"],
    hue=raw_data_cleaned["Cluster"],
    palette="tab10",
)
plt.title("Clusters of Vehicles (KMeans)")
plt.show()
```



KMeans Clustering Results

Feature Engineering

- Derived Features:
    - Vehicle Age: Extracted from the First Registration column.
    - Mileage per Year: Normalized mileage based on vehicle age.
- Encoded Features:
    - Categorical columns (Make, Fuel, Transmission, Drive type) were one-hot encoded.

    - Other categorical columns containing a high percentage of distinct values which are not relevant for the analysis (Interior Color, for example, we dropped)
- Transformed Features:
    - Log transformations applied to price (Log Price) and Mileage (Log Mileage) to handle skewness.
- Standardized Features:
    - Key numeric features (Mileage, Power, Cubic Capacity, Vehicle Age, Mileage per Year) were standardized to ensure consistent scaling.

```
# Derive Features
current_year = datetime.now().year
raw_data_cleaned["First Registration Year"] = (
    raw_data_cleaned["First Registration"].str.extract(r"(\d{4})").astype(float)
)
raw_data_cleaned["Vehicle Age"] = (
    current_year - raw_data_cleaned["First Registration Year"]
)
raw_data_cleaned["Mileage per Year"] = (
    raw_data_cleaned["Mileage"] / raw_data_cleaned["Vehicle Age"]
)

# Encoding Categorical Variables
categorical_columns_to_encode = [
    "Vehicle condition",
    "Make",
    "Category",
    "Door Count",
    "Emission Class",
    "Climatisation",
    "Parking sensors",
    "Airbags",
    "Colour",
    "Fuel",
    "Transmission",
]

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

# Drop all remaining categorical columns
categorical_columns_to_drop = raw_data_cleaned.select_dtypes(include="object").columns
raw_data_encoded = pd.concat(
    [raw_data_cleaned.drop(columns=categorical_columns_to_drop), encoded_columns],
    axis=1,
)

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

# Step 4: Standardization
numeric_features = [
```

```
    "Mileage",
    "Power",
    "Cubic Capacity",
    "Vehicle Age",
    "Mileage per Year",
]
scaler = StandardScaler()
raw_data_encoded[numeric_features] = scaler.fit_transform(
    raw_data_encoded[numeric_features]
)

# Step 5: Drop all rows with missing values
raw_data_encoded = raw_data_encoded.dropna()

print("Feature Engineering Summary:")
print(
    f"- Encoded Features: {encoder.get_feature_names_out(categorical_columns_to_encode).tolist()}"
)
print(f"- Dropped Columns: {categorical_columns_to_drop.tolist()}")
print("Transformed Dataset:")
print(raw_data_encoded.info())
```

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).

```
# Load data
data_df_encoded = pd.read_csv("./data/processed_data.csv")
X = data_df_encoded.drop(columns=["price", "Log Price"])
y = data_df_encoded["Log Price"]

# Train/test split
X, y = shuffle(X, y, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Dimensionality Reduction

We use 3 dimensionality reduction methods:

- PCA
- TruncatedSVD
- UMAP

All of our models will be trained using the features selected by each one of these methods and we will then compare the results

```
# Standardize before PCA
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# PCA: keep enough components to explain 95% variance
pca = PCA(n_components=0.95, random_state=42)
X_train_pca = pca.fit_transform(X_train_scaled)
```

```
X_test_pca = pca.transform(X_test_scaled)

print(f"PCA: {X_train_pca.shape[1]} components explain 95% variance.")

svd = TruncatedSVD(n_components=30, random_state=42)
X_train_svd = svd.fit_transform(X_train)
X_test_svd = svd.transform(X_test)
print(f"TruncatedSVD: {X_train_svd.shape[1]} components used.")

umap_reducer = umap.UMAP(n_components=20, random_state=42)
X_train_umap = umap_reducer.fit_transform(X_train)
X_test_umap = umap_reducer.transform(X_test)
print(f"UMAP: {X_train_umap.shape[1]} components used.")
```

Model Training

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

- Linear Regression
- Ridge Regression
- Lasso Regression
- Random Forest Regressor
- SVR (Support Vector Regression)
- CatBoostRegressor (Categorical Gradient Boosting)
- XGBoost (Extreme Gradient Boosting)
- LightGBM

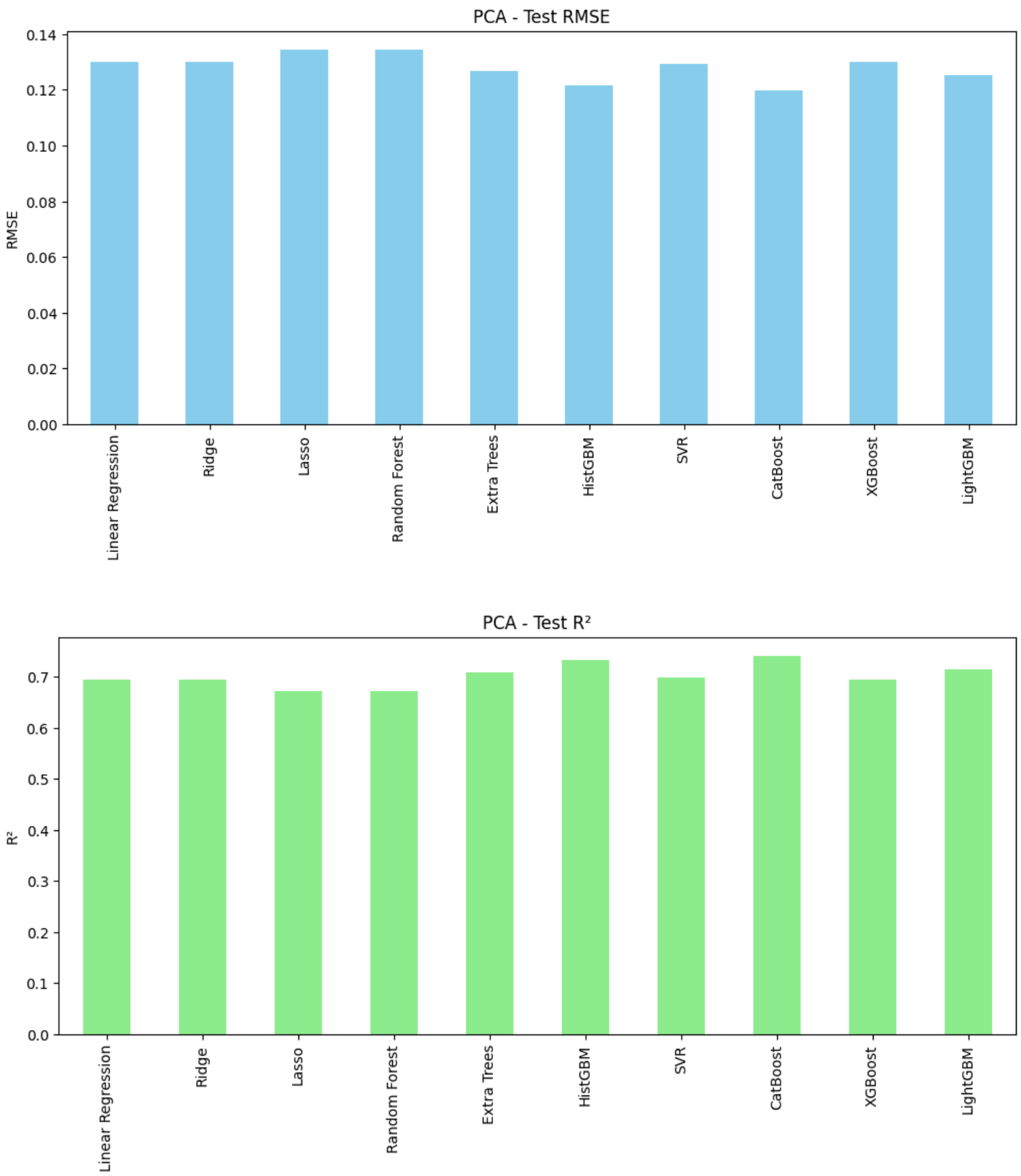
First, we train the models using set hyperparameters to get a baseline for their performance. We then evaluate their Root Mean Squared Error, Mean Absolute Error and their R^2.

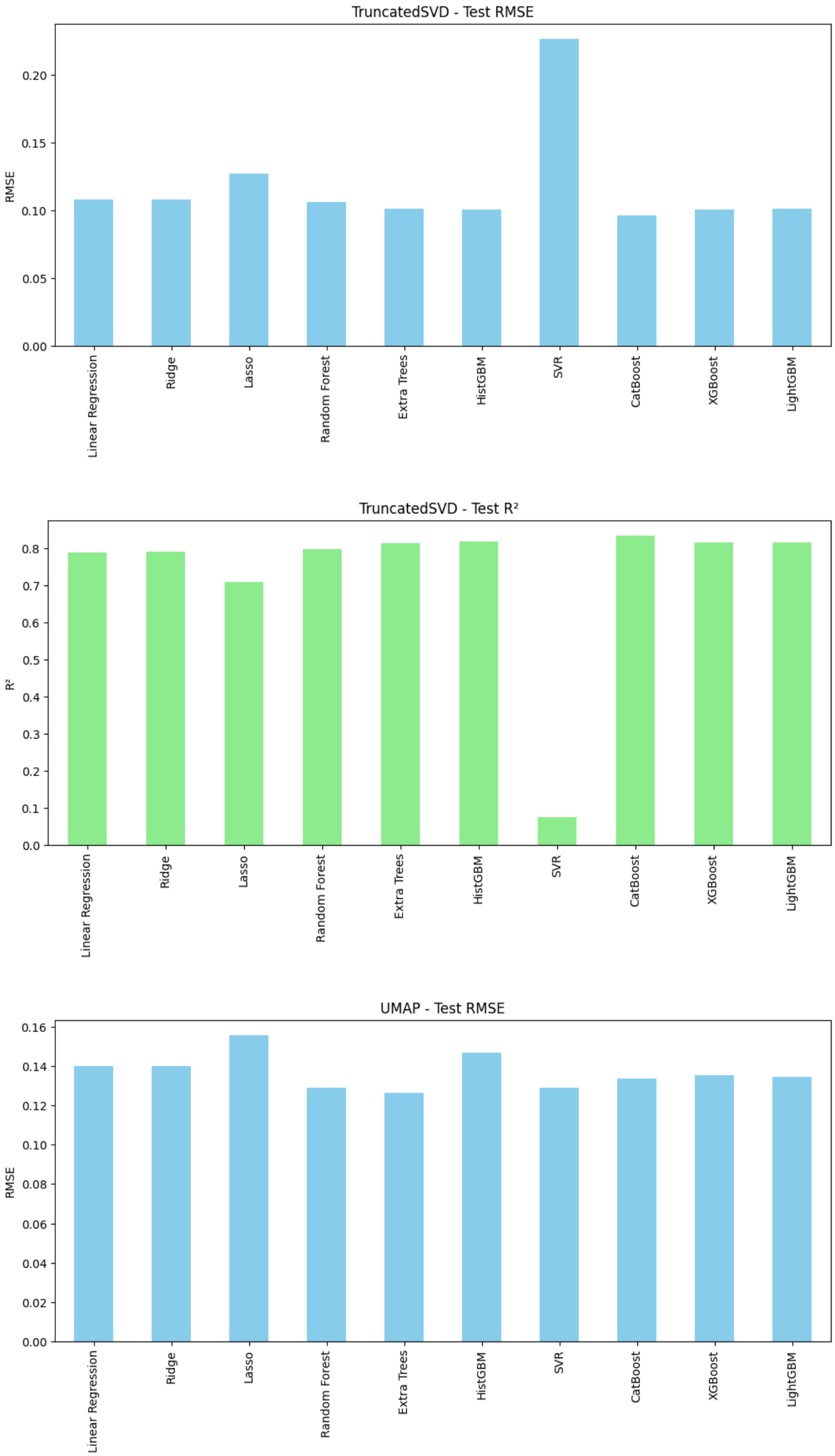
```
# Train Models (Without Hyperparameter Tuning) with Selected Features to get a sense of performance
models = {
    "Linear Regression": LinearRegression(),
    "Ridge": Ridge(alpha=1.0),
    "Lasso": Lasso(alpha=0.01, max_iter=10000),
    "Random Forest": RandomForestRegressor(
        random_state=42, n_estimators=200, max_depth=15
    ),
    "Extra Trees": ExtraTreesRegressor(random_state=42, n_estimators=200, max_depth=15),
    "HistGBM": HistGradientBoostingRegressor(random_state=42, max_iter=200),
    "SVR": SVR(kernel="rbf", C=10, gamma="scale"),
    "CatBoost": CatBoostRegressor(
        verbose=0, random_state=42, learning_rate=0.05, iterations=500, depth=4
    ),
    "XGBoost": XGBRegressor(
        random_state=42, n_estimators=200, max_depth=5, learning_rate=0.05
    ),
    "LightGBM": LGBMRegressor(
        random_state=42, n_estimators=200, max_depth=5, learning_rate=0.05, verbose=-1
    ),
}

def evaluate_models(X_tr, X_te, y_tr, y_te, models, title=""):
    results = {}
    for name, model in models.items():
```

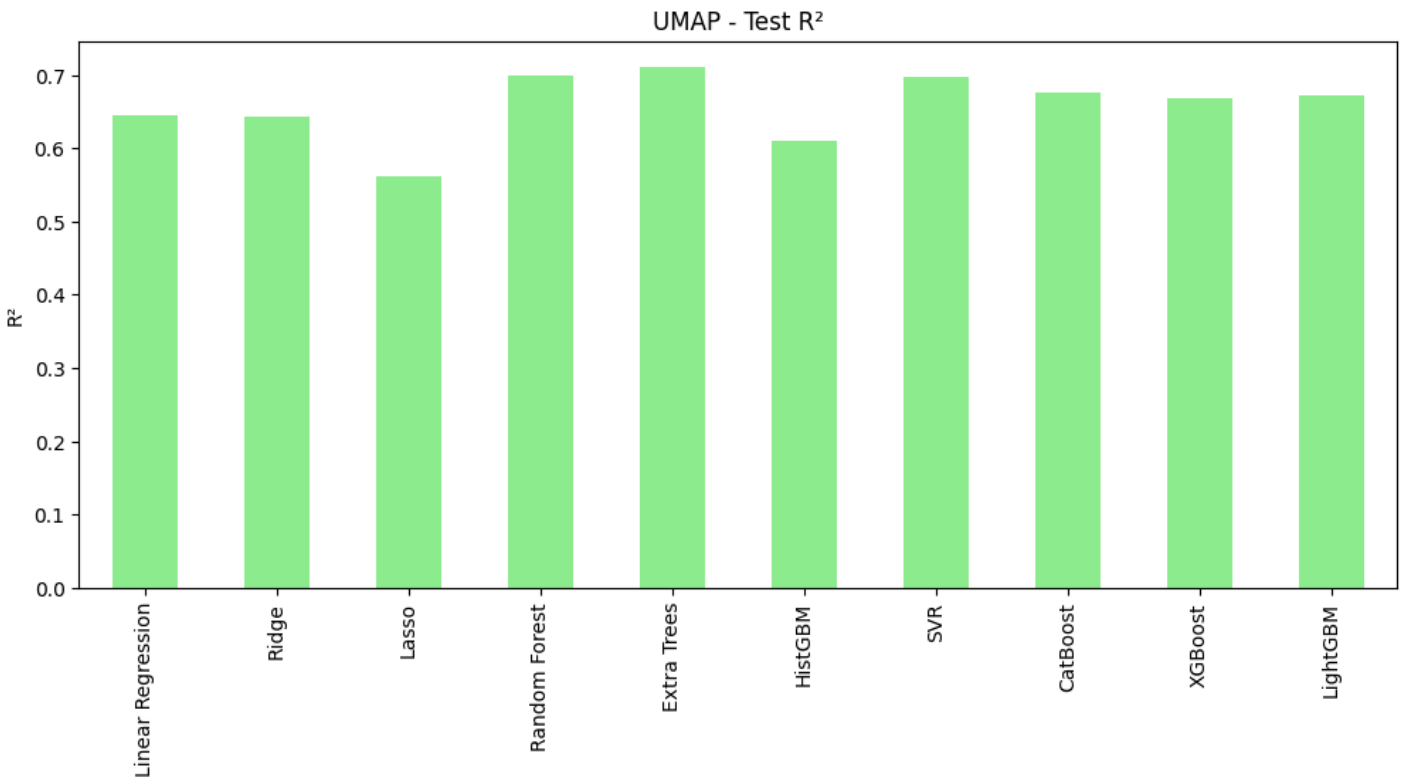
```
        model.fit(X_tr, y_tr)
        y_pred = model.predict(X_te)
        y_pred_train = model.predict(X_tr)
        results[name] = {
            "RMSE Test": np.sqrt(mean_squared_error(y_te, y_pred)),
            "MAE Test": mean_absolute_error(y_te, y_pred),
            "R2 Test": r2_score(y_te, y_pred),
            "RMSE Train": np.sqrt(mean_squared_error(y_tr, y_pred_train)),
            "R2 Train": r2_score(y_tr, y_pred_train),
        }
    results_df = pd.DataFrame(results).T
    print(f"\nResults ({title}):\n", results_df)
    return results_df


# Evaluate on PCA, SVD, UMAP
results_pca = evaluate_models(X_train_pca, X_test_pca, y_train, y_test, models, "PCA")
results_svd = evaluate_models(
    X_train_svd, X_test_svd, y_train, y_test, models, "TruncatedSVD"
)
results_umap = evaluate_models(
    X_train_umap, X_test_umap, y_train, y_test, models, "UMAP"
)
```

Model	Dimensionality Reduction Method	RMSE Test	MAE Test	R^2 Test	RMSE Train	R^2 Train
Linear Regression	PCA	0.130	0.100	0.694	0.119	0.774
Ridge	PCA	0.130	0.100	0.694	0.119	0.774
Lasso	PCA	0.134	0.102	0.673	0.134	0.715
Random Forest	PCA	0.134	0.104	0.673	0.055	0.951
Extra Trees	PCA	0.127	0.099	0.709	0.012	0.998
HistGBM	PCA	0.122	0.095	0.732	0.007	0.999
SVR	PCA	0.129	0.099	0.697	0.077	0.906
CatBoost	PCA	0.120	0.092	0.740	0.061	0.940
XGBoost	PCA	0.130	0.100	0.694	0.027	0.988
LightGBM	PCA	0.125	0.098	0.715	0.040	0.974
Linear Regression	SVD	0.108	0.083	0.789	0.115	0.790
Ridge	SVD	0.108	0.082	0.790	0.115	0.790
Lasso	SVD	0.127	0.099	0.709	0.137	0.699
Random Forest	SVD	0.106	0.082	0.796	0.046	0.967
Extra Trees	SVD	0.101	0.079	0.814	0.009	0.999
HistGBM	SVD	0.101	0.075	0.817	0.011	0.998
SVR	SVD	0.226	0.180	0.074	0.241	0.074
CatBoost	SVD	0.096	0.070	0.833	0.062	0.939
XGBoost	SVD	0.101	0.077	0.816	0.037	0.978
LightGBM	SVD	0.101	0.077	0.814	0.050	0.961
Linear Regression	UMAP	0.140	0.110	0.644	0.150	0.644
Ridge	UMAP	0.140	0.110	0.644	0.151	0.639
Lasso	UMAP	0.156	0.121	0.561	0.172	0.529
Random Forest	UMAP	0.129	0.102	0.699	0.051	0.958

Extra Trees	UMAP	0.126	0.100	0.711	0.013	0.997
HistGBM	UMAP	0.147	0.115	0.610	0.039	0.976
SVR	UMAP	0.129	0.103	0.697	0.137	0.703
CatBoost	UMAP	0.134	0.104	0.675	0.099	0.843
XGBoost	UMAP	0.135	0.107	0.669	0.072	0.917
LightGBM	UMAP	0.135	0.105	0.672	0.092	0.864




TruncatedSVD - Test RMSE
RMSE
0.00
0.05
0.10
0.15
0.20
Linear Regression
Ridge
Lasso
Random Forest
Extra Trees
HistGBM
SVR
CatBoost
XGBoost
LightGBM
TruncatedSVD - Test R²
R²
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Linear Regression
Ridge
Lasso
Random Forest
Extra Trees
HistGBM
SVR
CatBoost
XGBoost
LightGBM
UMAP - Test RMSE
RMSE
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
Linear Regression
Ridge
Lasso
Random Forest
Extra Trees
HistGBM
SVR
CatBoost
XGBoost
LightGBM



Baseline Models Evaluation

- For most of our models, the dataset on which we used the features selected by the TruncatedSVD method seem to give the best results apart from the SVR model which performs poorly, so it will need hyperparameter tuning
- The linear models generally perform quite well using all dimensionality reduction methods, though not as well as the Gradient Boosting models

Hyperparameter Tuning

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

```
models_and_grids = {
    "Random Forest": (
        RandomForestRegressor(random_state=42),
        {"n_estimators": [100, 200], "max_depth": [10, 20]},
    ),
    "Extra Trees": (
        ExtraTreesRegressor(random_state=42),
        {"n_estimators": [100, 200], "max_depth": [10, 20]},
    ),
    "HistGBM": (
        HistGradientBoostingRegressor(random_state=42),
        {"max_iter": [100, 200], "max_depth": [None, 10]},
    ),
    "SVR": (SVR(), {"C": [1, 10], "gamma": ["scale", "auto"]}),
    "Ridge": (Ridge(), {"alpha": [0.1, 1, 10]}),
    "Lasso": (Lasso(max_iter=10000), {"alpha": [0.01, 0.1, 1]}),
    "CatBoost": (
        CatBoostRegressor(verbose=False, random_state=42),
        {"iterations": [500], "learning_rate": [0.01, 0.05], "depth": [4, 6]},
    ),
    "XGBoost": (
        XGBRegressor(random_state=42),
        {
            "n_estimators": [100, 200],
            "max_depth": [3, 5],
            "learning_rate": [0.01, 0.05],
        },
    ),
    "LightGBM": (
```

```
        LGBMRegressor(verbose=-1, random_state=42),
        {
            "n_estimators": [100, 200],
            "max_depth": [3, 5],
            "learning_rate": [0.01, 0.05],
        },
    ),
}
```

```
reduced_sets = {
    "PCA": (X_train_pca, X_test_pca),
    "SVD": (X_train_svd, X_test_svd),
    "UMAP": (X_train_umap, X_test_umap),
}
```

```
all_results = {}

for red_name, (Xtr, Xte) in reduced_sets.items():
    print(f"\n--- {red_name} ---")
    results = {}
    for model_name, (model, param_grid) in models_and_grids.items():
        print(f"Tuning {model_name}...")
        grid = GridSearchCV(
            model, param_grid, scoring="neg_root_mean_squared_error", cv=3, n_jobs=-1
        )
        grid.fit(Xtr, y_train)
        best = grid.best_estimator_
        y_pred = best.predict(Xte)
        y_pred_train = best.predict(Xtr)
        results[model_name] = {
            "Best Params": grid.best_params_,
            "RMSE Test": np.sqrt(mean_squared_error(y_test, y_pred)),
            "MAE Test": mean_absolute_error(y_test, y_pred),
            "R2 Test": r2_score(y_test, y_pred),
            "RMSE Train": np.sqrt(mean_squared_error(y_train, y_pred_train)),
            "R2 Train": r2_score(y_train, y_pred_train),
        }
    results_df = pd.DataFrame(results).T
    results_df["Reduction Method"] = red_name
    results_df["Model"] = results_df.index
    print(results_df[["RMSE Test", "MAE Test", "R2 Test", "Best Params"]])
    all_results[red_name] = results_df
```

Selected Best Parameters

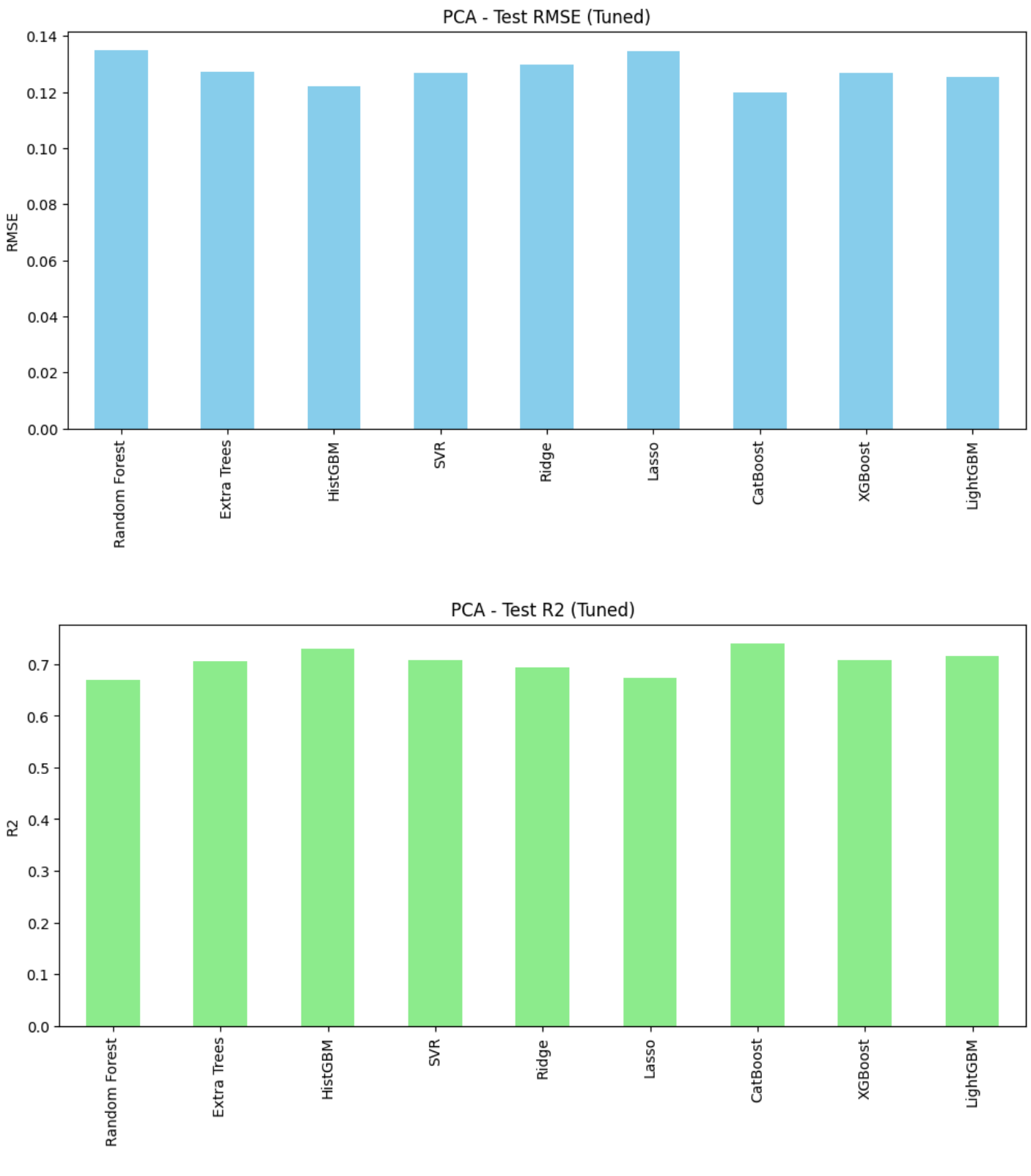
Model	Reduction Method	Best Params
Random Forest	PCA	{'max_depth': 20, 'n_estimators': 200}
Extra Trees	PCA	{'max_depth': 20, 'n_estimators': 200}
HistGBM	PCA	{'max_depth': 10, 'max_iter': 200}
SVR	PCA	{'C': 1, 'gamma': 'scale'}
Ridge	PCA	{'alpha': 10}
Lasso	PCA	{'alpha': 0.01}
CatBoost	PCA	{'depth': 4, 'iterations': 500, 'learning_rate': 0.05}
XGBoost	PCA	{'learning_rate': 0.05, 'max_depth': 3, 'n_estimators': 200}

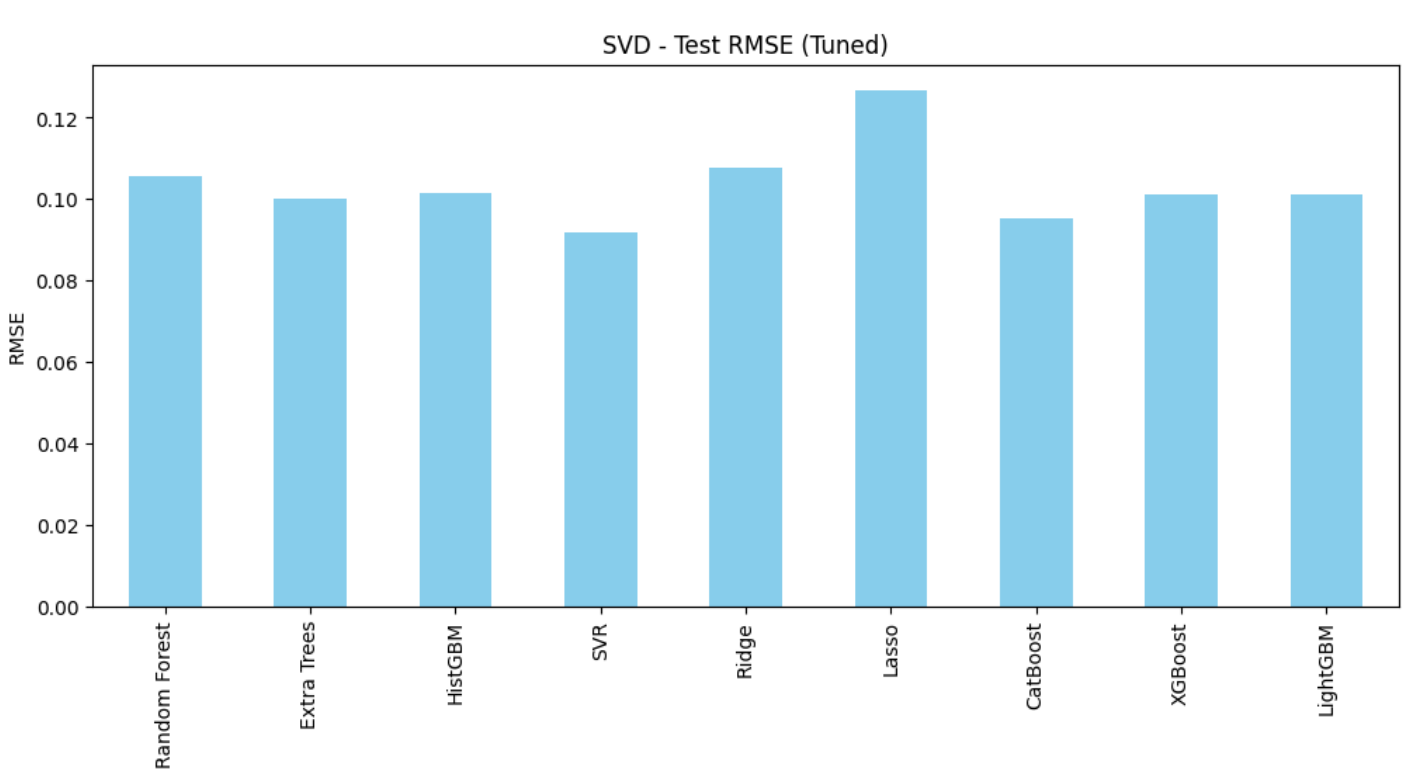
Model	Reduction Method	Best Params
LightGBM	PCA	{'learning_rate': 0.05, 'max_depth': 5, 'n_estimators': 200}
Random Forest	SVD	{'max_depth': 20, 'n_estimators': 200}
Extra Trees	SVD	{'max_depth': 20, 'n_estimators': 200}
HistGBM	SVD	{'max_depth': 10, 'max_iter': 200}
SVR	SVD	{'C': 1, 'gamma': 'auto'}
Ridge	SVD	{'alpha': 1}
Lasso	SVD	{'alpha': 0.01}
CatBoost	SVD	{'depth': 6, 'iterations': 500, 'learning_rate': 0.05}
XGBoost	SVD	{'learning_rate': 0.05, 'max_depth': 3, 'n_estimators': 200}
LightGBM	SVD	{'learning_rate': 0.05, 'max_depth': 5, 'n_estimators': 200}
Random Forest	UMAP	{'max_depth': 10, 'n_estimators': 200}
Extra Trees	UMAP	{'max_depth': 10, 'n_estimators': 200}
HistGBM	UMAP	{'max_depth': 10, 'max_iter': 100}
SVR	UMAP	{'C': 10, 'gamma': 'auto'}
Ridge	UMAP	{'alpha': 0.1}
Lasso	UMAP	{'alpha': 0.01}
CatBoost	UMAP	{'depth': 6, 'iterations': 500, 'learning_rate': 0.05}
XGBoost	UMAP	{'learning_rate': 0.05, 'max_depth': 5, 'n_estimators': 100}
LightGBM	UMAP	{'learning_rate': 0.05, 'max_depth': 5, 'n_estimators': 100}

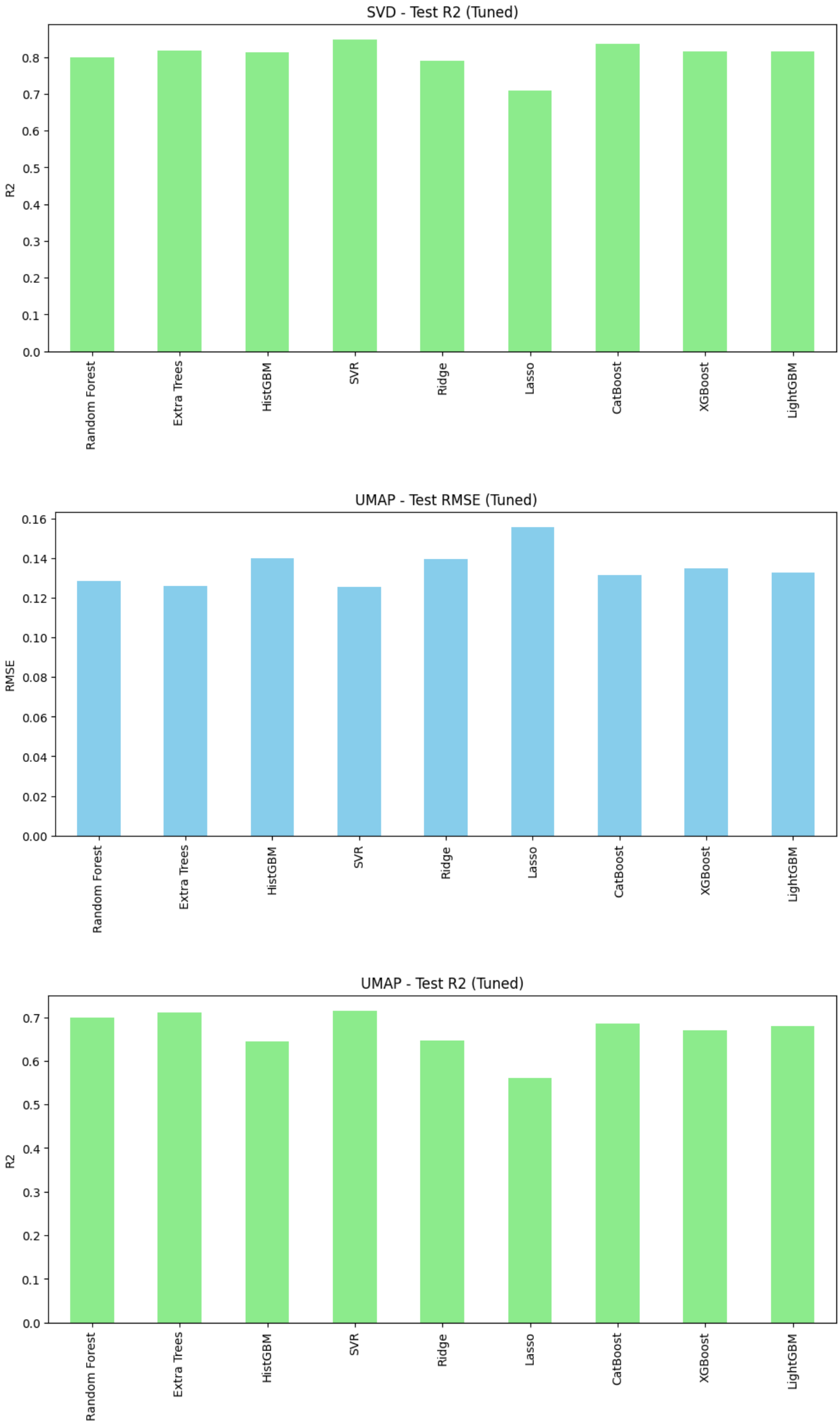
Results of Hyperparameter Tuning

Model	Reduction Method	RMSE Test	MAE Test	R2 Test	RMSE Train	R2 Train
Random Forest	PCA	0.13493163	0.10468231	0.6702082	0.05488451	0.95202736
Extra Trees	PCA	0.12731302	0.09913038	0.70639871	0.00102133	0.99998339
HistGBM	PCA	0.1220706	0.09468744	0.73008037	0.0064491	0.99933764
SVR	PCA	0.1268346	0.09674795	0.70860118	0.07972777	0.89876905
Ridge	PCA	0.1298749	0.10020187	0.69446377	0.1190165	0.77441597
Lasso	PCA	0.13438343	0.10173066	0.67288251	0.13388391	0.71453636
CatBoost	PCA	0.1197726	0.09175165	0.74014729	0.06134297	0.94007286
XGBoost	PCA	0.12692844	0.09768546	0.70816983	0.08007688	0.89788059
LightGBM	PCA	0.12536968	0.09797733	0.71529351	0.04037675	0.97403689
Random Forest	SVD	0.10548205	0.08161044	0.79845619	0.04544128	0.9671152
Extra Trees	SVD	0.10002096	0.07818997	0.81878491	0.0008348	0.9999889
HistGBM	SVD	0.10133811	0.07541121	0.81398075	0.01366206	0.99702747
SVR	SVD	0.09165542	0.06828434	0.84783014	0.07848576	0.90189847
Ridge	SVD	0.10779376	0.08240228	0.78952547	0.11490115	0.78974671
Lasso	SVD	0.12672512	0.09941714	0.709104	0.13737229	0.69946689
CatBoost	SVD	0.09538088	0.07116632	0.83520846	0.03944795	0.97521763
XGBoost	SVD	0.10109236	0.07869966	0.81488185	0.07510524	0.9101673
LightGBM	SVD	0.1012397	0.07712901	0.81434187	0.04966925	0.96071116
Random Forest	UMAP	0.1286886	0.10198047	0.70001988	0.06805475	0.92624169
Extra Trees	UMAP	0.1261317	0.0988088	0.71182201	0.06008026	0.94251458
HistGBM	UMAP	0.14010725	0.1104788	0.64442311	0.071579	0.91840467
SVR	UMAP	0.12546342	0.09855347	0.71486761	0.13073753	0.72779597
Ridge	UMAP	0.13963314	0.10929851	0.64682552	0.14967186	0.64324151
Lasso	UMAP	0.1556868	0.12102138	0.56094805	0.17197906	0.5289737

Model	Reduction Method	RMSE Test	MAE Test	R2 Test	RMSE Train	R2 Train
CatBoost	UMAP	0.13164284	0.10246786	0.6860888	0.08228142	0.89218043
XGBoost	UMAP	0.13488683	0.10604233	0.67042718	0.08578054	0.88281513
LightGBM	UMAP	0.13283483	0.1046026	0.68037833	0.10181261	0.83491899







Conclusions

After analyzing the performance of the various models with different dimensionality reduction techniques, the findings were as follows:

Best Overall Performance:

- SVR with TruncatedSVD features achieved the best results with:

- RMSE Test: 0.092
- MAE Test: 0.068
- R^2 Test: 0.848

Top Performing Models:

- CatBoost with TruncatedSVD features was the second-best performer, followed by Extra Treesand HistGBM

Dimensionality Reduction Methods:

- TruncatedSVD consistently outperformed both PCA and UMAP across most models

Model Types:

- Gradient Boosting models (CatBoost, HistGBM) and tree-based models (Random Forest, Extra Trees) generally performed better than linear models