CS5344: Big Data Analytics

Final Project Presentation

By
Adithya Ragothaman,
A0274780W.
Team-9

Motivation

Project Goal:

Create a machine learning (regression) model to predict vehicle prices based on key factors such as make, model, year, mileage, condition, and location.

Industry Context:

The automobile industry is highly competitive, with millions of vehicle transactions annually.
Understanding the fair market value is essential for buyers and sellers alike.

Business Challenge:

Determining the fair market value of a vehicle is complex due to many influencing factors, making it difficult for consumers, dealers, and manufacturers to make informed decisions.

Target task

Target Task:

The target task is to **predict the price of a vehicle** based on various features such as mileage, age, brand, days on the market, dealer type, engine specifications, and other vehicle attributes. By using these features, the Regression ML models aims to capture the key factors influencing vehicle resale value, providing accurate price predictions to support informed decision-making in vehicle purchasing and sales.

Regression Models used for price prediction:

1. Linear Regression
2. Decision Trees
3. Random Forest
4. GBT
5. XGB

KPI: (Metrics to evaluate regression model's performance)

1. R2 Score
2. MAE
3. MAPE
4. RMSE

Note:

Since this is a pricing model, both MAE and RMSE are expressed in dollars, providing a direct measure of the average and root mean square errors in terms of price.

Dataset Overview

Dataset taken from : Kaggle/us-used-cars-dataset

Key Features:

Vehicle Information:

Columns like model_name (model of the vehicle), and year (manufacturing year).

Physical Attributes:

Features such as engine_displacement, and fuel_type provide technical specifications that can influence vehicle price.

Market & Seller Details:

Variables like dealer_zip (location of the dealer), seller_rating (ratings given to the seller), and daysonmarket (how long the vehicle has been for sale) are included to assess market trends.

Numerical and Categorical Data:

The dataset has a mix of numerical features (e.g., mileage,horsepower, fuel_tank_volume) and categorical features (e.g., model_name, fuel_type, body_type), providing a **rich dataset for machine learning models.**

Target Variable:

The price column is used as the target variable for prediction, representing the final car sale price.

Big Data Relevance:

Data Volume:

The dataset includes **3 million vehicle listings.**

Handling such a large volume of data presents challenges in data storage, processing, and analysis.

Variety of Data:

The dataset contains a variety of features from complex vehicle features and simple features like color. This variety adds complexity, requiring sophisticated feature engineering & data preprocessing.

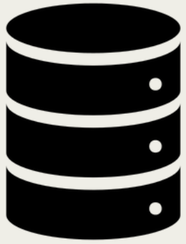
Tools and Techniques:

Big Data technology : **PySpark** is used completely in the project for and large scale and parallel processing and prediction.

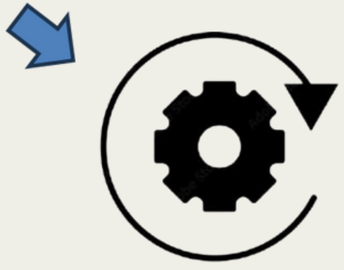
Challenges in Big Data:

Along with handling data volume, variety, and velocity, the project also faces challenges related to data cleaning, missing values, and selecting the right machine learning algorithms to ensure scalability and accuracy. One more Major Factor is Time taken to train the models.

Overall Project Workflow

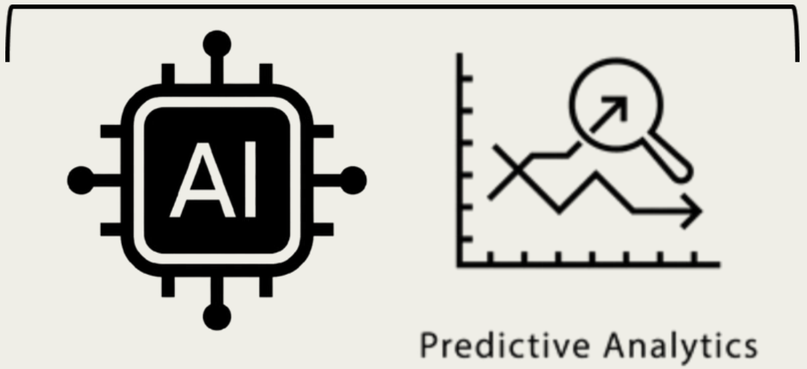


Dataset stored as CSV

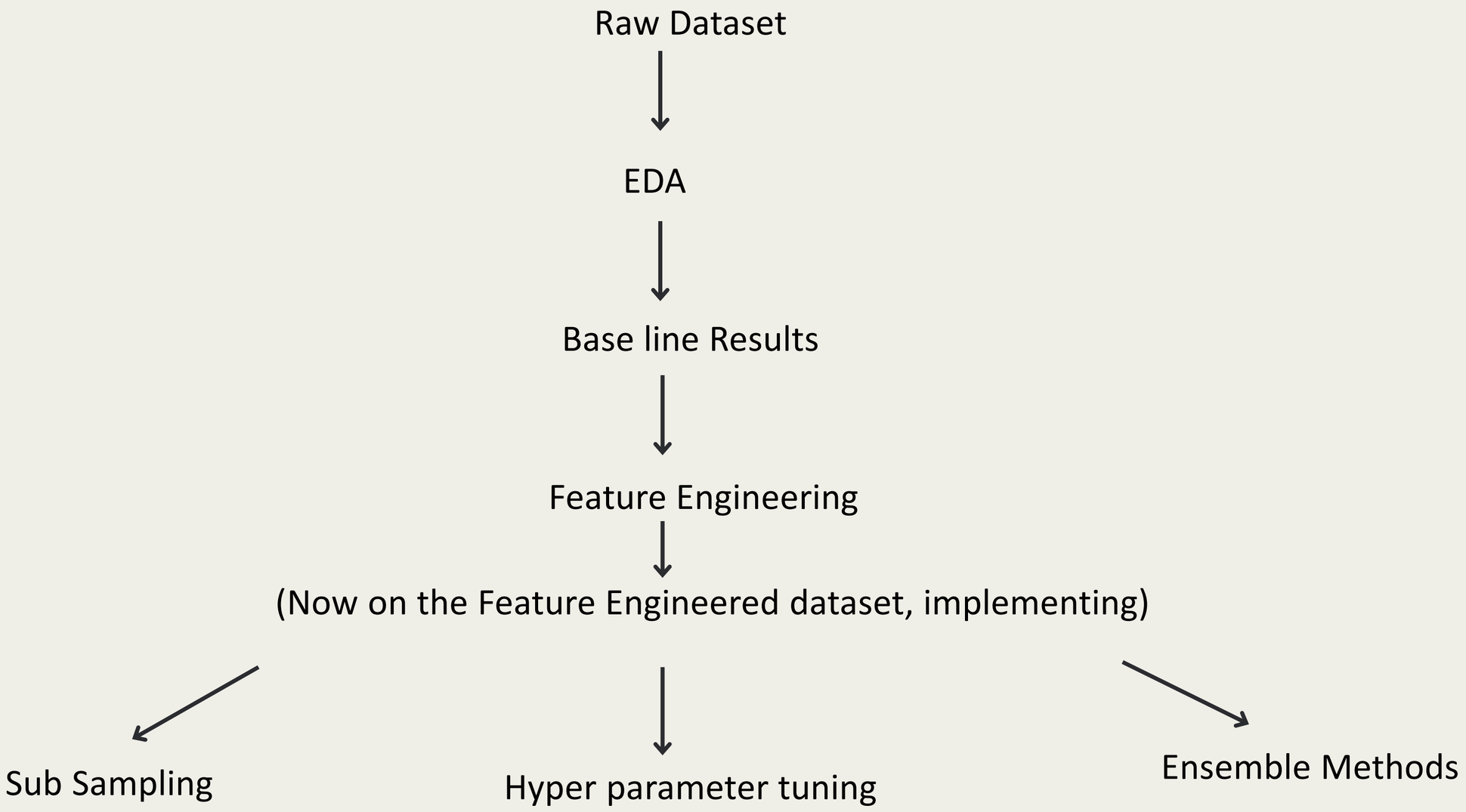


Processed Dataset stored as Parquet file after EDA

Price Prediction Using Spark ML Algorithms -> Multiple iterations, Different Approaches



Detailed Technical Workflow



Main Approaches and Experimental Techniques

Main Approach:

- Data Preparation: Raw Dataset ➔ **EDA** ➔ Baseline Results
- Feature Engineering on the processed dataset
- Ensemble Method:
  - Bagging (XGBoost, Decision Tree)

Experiments:

- Hyperparameter Tuning for model optimization
- Sub-Sampling Techniques:
  a. Row-wise sampling
  b. Column-wise sampling
- BERT Embeddings for Description Column: Feature Transformation

Approaches

EDA

Column-wise EDA

Dataset Size before EDA

DataFrame has 3000040 rows and 66 columns

Dataset Size After EDA

DataFrame has 3000040 rows and 42 columns

Columns deleted: **27**

Columns added: **5**

New columns:

1. major_options_count
2. log_mileage
3. combined_fuel_economy
4. legroom
5. RPM

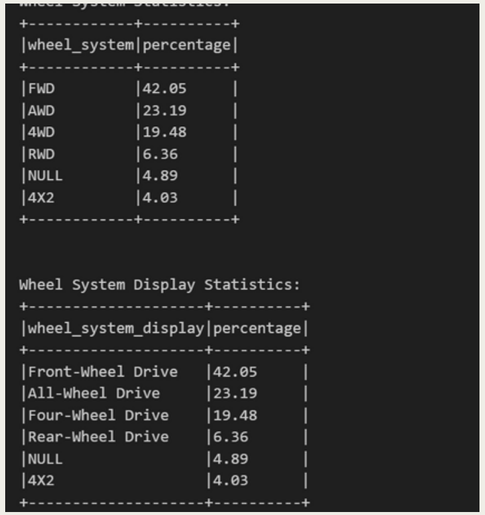
Row-wise EDA

1. Numerical Columns Handling:
   Null values in numerical columns were replaced with central tendency measures: the **mean** or **median**, depending on the nature of the data. In some cases, the **mode** was used for filling missing values.

2. Categorical Columns Handling:
   For categorical columns, null values were either replaced with the most frequent category (**mode**) or assigned the value '**Unknown**' to maintain consistency in the dataset.

Columnar EDA

Null values : Bin	No. of Columns
0-10%	44
10-20%	6
40-50%	7
80-90%	2
90-100%	7



```
+------------+----------+
|wheel_system|percentage|
+------------+----------+
|FWD         |42.05     |
|AWD         |23.19     |
|4WD         |19.48     |
|RWD         |6.36      |
|NULL        |4.89      |
|4X2         |4.03      |
+------------+----------+


Wheel System Display Statistics:
+--------------------+----------+
|wheel_system_display|percentage|
+--------------------+----------+
|Front-Wheel Drive   |42.05     |
|All-Wheel Drive     |23.19     |
|Four-Wheel Drive    |19.48     |
|Rear-Wheel Drive    |6.36      |
|NULL                |4.89      |
|4X2                 |4.03      |
+--------------------+----------+
```

Actions

1. Dropped the columns which had more than 40% null values: 16 columns.

2. Dropped due to similarity
   a. **engine_cylinders:** Similar to 'engine_type', so removed for redundancy.
   b. **franchise_make**: Dropped due to overlap with 'make_name', which is more complete.
   c. **wheel_system:** Removed as 'wheel_system_display' provides the same info more clearly.
   d. **power:** Redundant with 'horsepower', so deleted.

3. Trimming Unnecessary Data.
   a. ID Columns: Dropped 'vin', 'listing_id', 'sp_id', 'trimId'.
   b. URL Column: Removed 'main_picture_url'.
   c. trim_name: Dropped as unnecessary.

4. Processed the following columns from string format to numerical (e.g., '35.1 in' -> 35.1):
   wheelbase, width,fuel_tank_volume, height, maximum_seating, and length, etc....

5. Combined Columns:
   a. ‘city_fuel_economy’ + ‘highway_fuel_economy’= **combined_fuel_economy**
   b. ‘back_legroom’ + ‘front_legroom’ = **legroom**

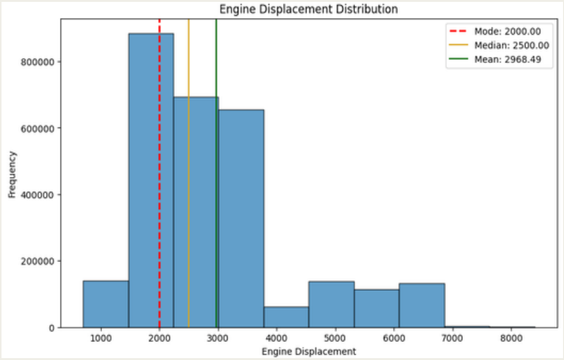
6. Split the torque column into two new columns: torque and RPM.
   Did this to separately capture the torque value and its RPM for better feature representation.

7. Added ‘**major_options_count**’ and ‘**log_mileage**’ as mileage was unevenly distributed.

Row EDA

Number of rows with at least one null value: 1119596
(1/3rd of Data)

I plotted the graph for all these columns and decided whether to replace by median/mode/mean for numerical columns



Categorical Columns with Non-Zero Null Percentages:

Categorical_Columns	Null_Percentage
major_options	6.668
wheel_system_display	4.891
engine_type	3.353
fuel_type	2.757
description	2.597
transmission	2.139
transmission_display	2.139
body_type	0.451
interior_color	0.000

Numerical Columns with Non-Zero Null Percentages:

Numerical_Columns	Null_Percentage
torque	17.260
city_fuel_economy	16.376
highway_fuel_economy	16.376
combined_fuel_economy	16.376
legroom	8.091
engine_displacement	5.746
horsepower	5.746
fuel_tank_volume	5.356
maximum_seating	5.326
width	5.325
height	5.324
length	5.324
wheelbase	5.323
mileage	4.813
seller_rating	1.362

1. Columns where null values are replaced by **mean**:
    a. Torque
    b. Fuel Economy (City, Highway, Combined)
    c. Fuel Tank Volume (mean by respective engine type)
    d. Height

2. Columns where null values are replaced by **median**:
    a. Length
    b. Width
    c. Horsepower
    d. Mileage
    e. Log Mileage
    f. Wheelbase
    g. Engine Displacement

3. Columns where null values are replaced by **mode**:
    a. Maximum Seating
    b. Rating
    c. Categorical:
        - Transmission
        - Body Type

4. Categorical columns where null values are replaced by **'Unknown'**:
    a. Major Options
    b. Description
    c. Wheel System Display
    d. Transmission Display

Baseline Results

Model	R2 Score	Mean Average Error (MAE)	MAE as percentage of Mean (MAPE) (avg.price of car = 29,933 $)	RMSE	RMSE as percentage of Mean (avg.price of car = 29,933 $)	Total Hours Trained
Linear Regression	0.775	4284	14.30 %	9170	30.63 %	7 hours
Decision Tree Regressor	0.827	3265	~ 11.00 %	8038	26.85 %	1.0 hours
Random Forest	0.781	3791	12.66 %	9164	30.61 %	4.0 hours
XGBoost	0.854	3066	10.24 %	7515	25.10 %	2 hours
GBT Regressor	0.832	3578	11.95 %	7943	26.53 %	~ 4.0 hours

Before the Model training, I am

1. df=df.drop('description','major_options','mileage').
2. Processes the exterior and interior color columns by identifying known colors, labeling multiple colors as "Mixed Colors," and assigning "Other" for unrecognized or missing values.

Proposed methods to **Beat the Baseline**

Proposal to implement the following to improve the baseline model:

1. Feature Engineering

- Generated new features to capture relationships in data
- Extracted time-related attributes
- Incorporated domain-specific insights

2. Hyper parameter tuning

- Conducted hyperparameter tuning for each of the 5 models
- Tested with 8-16 different parameters per model

3. Ensemble methods

- **Bagging :**
  - combines predictions from multiple models trained on different data samples.
  - Implemented bagging using XGBoost and Decision Tree models.
    (Gave me the BEST results: reduced error to less than 4%)

4. Sub Sampling

- **Row Sampling:**
  - Evaluated model performance with varying data sizes to understand the impact of increased rows.
  - Trained models on samples of 100k, 200k, 300k, and 600k rows across all five models.
- **Column Sampling**
  - For each model, ran 10 iterations with 70% randomly selected features on each iteration.
  - Model trained: XGB Regressor and Decision Tree

Feature Engineering

New columns:

1.) Interaction Features:

Create interaction terms between existing features to capture relationships between variables.

- hp_x_engine_disp
- hp_x_torque

2.) Date-based Features:

Extract new features from dates such as age or year-on-year trends to understand the time-dependent effects on price.

- listed_day
- listed_month
- listed_year
- Age

3.) Domain-Specific Feature Engineering:

I have used vehicle-specific domain knowledge to create meaningful features.

- resale_value_score
- maintenance_cost
- luxury_score

Feature Engineering

Interaction Features:

Step 1: Calculating Correlations

I calculated the correlation matrix for numerical columns in the dataset to identify highly correlated pairs to find relationships between features, such as horsepower and engine_displacement
(0.83 correlation) and horsepower and torque (0.79 correlation).

Step 2: Filtering High Correlation Pairs

I filtered feature pairs with correlation values greater than 0.75, indicating strong relationships.
These 2 pairs of columns were identified as potential candidates for interaction terms, because of their unique synergy.

Step 3: Creating Interaction Terms

I created two interaction terms: hp_x_engine_disp (horsepower × engine_displacement) and hp_x_torque (horsepower × torque).

```
Column pairs with correlation > 0.75 or < -0.75:

combined_fuel_economy:
    city_fuel_economy                0.986976

highway_fuel_economy:
    combined_fuel_economy            0.983321

wheelbase:
    length                           0.966102

highway_fuel_economy:
    city_fuel_economy                0.941255

horsepower:
    engine_displacement              0.828887

fuel_tank_volume:
    length                           0.815081

fuel_tank_volume:
    wheelbase                        0.811318

horsepower:
    torque                           0.793814

fuel_tank_volume:
    engine_displacement              0.785700

engine_displacement:
    length                           0.779008

engine_displacement:
    wheelbase                        0.759748

year:
    mileage                         -0.751842

is_new:
    log_mileage                     -0.810650

longitude:
    dealer_zip                      -0.920972

listed_date:
    daysonmarket                    -0.984415
```

Domain-Specific Feature Engineering:

resale_value_score

This gives an overall indication of how well a vehicle is expected to retain its value based on these key factors.
resale_value_score = (‘Mileage Score’ + ‘Age Score’ + ‘Days on Market Scor’e + ‘Make Name Score’) **/ 4**

maintenance_cost

I calculate the total maintenance cost by summing several factors that influence a vehicle's maintenance.
maintenance_cost = (‘Brand Cost’ + ‘Age Cost’ + ‘Horsepower Cost’ + ‘Transmission Cost’ + ‘Fuel Type Cost’+ ‘Engine Type Cost’) **/ 6**

luxury_score

I calculate the overall luxury level of a vehicle by summing various factors that reflect the premium features and exclusivity of the vehicle.
luxury_score = (‘Brand Cost’+ ‘Manufacturing Year Score’ + ‘Seating Score’ + ‘Options Score’ + ‘Transmission Score’) **/ 5**

Example

```python
# UDF for mileage score related to resale value
def mileage_score_udf(log_mileage):
    if log_mileage < 2.3:                # Very low mileage (below 25th percentile), highest resale value
        return 10
    elif log_mileage < 8.91:             # Moderately low mileage (between 25th percentile and median), good resale value
        return 7
    elif log_mileage < 10.63:            # Moderate mileage (between median and 75th percentile), average resale value
        return 4
    else:                                # High mileage (above 75th percentile), lower resale value
        return 1

# UDF for age score related to resale value
def age_score_udf(age):                  # Very new cars (2 years old or less), highest resale value
    if age <= 2:
        return 10
    elif age <= 5:                       # Relatively new cars (3 to 5 years old), still good resale value
        return 7
    elif age <= 10:                      # Moderately aged cars (6 to 10 years old), average resale value
        return 4
    else:                                # Older cars (more than 10 years old), lower resale value
        return 1

# UDF for days on market score related to resale value (Assumes that vehicles with fewer days on the market have higher demand and higher resale value)
def days_in_market_score_udf(days_in_market):
    if days_in_market < 14:              # Very high demand (quick sale within 2 weeks), highest resale value
        return 10
    elif days_in_market < 35:            # High demand (sold within 1 month), good resale value
        return 7
    elif days_in_market < 82:            # Average demand (sold within 3 months), moderate resale value
        return 5
    elif days_in_market < 215:           # Lower demand (sold within 6-7 months), lower resale value
        return 3
    else:                                # Very low demand (vehicle has been on the market for over 7 months), lowest resale value
        return 1

# UDF for make_name score related to resale value using broadcasted brand lists (Vehicle make (brand) significantly affects resale value based on the brand's market position ).
def make_name_score_udf(make_name):
    if make_name in ultra_luxury_brands_bc.value:  # Ultra-luxury brands (e.g., Rolls-Royce, Bentley), highest resale value
        return 10
    elif make_name in luxury_brands_bc.value:      # Luxury brands (e.g., BMW, Volvo), very high resale value
        return 8
    elif make_name in high_end_brands_bc.value:    # High-end brands (e.g., Jeep, Lexus), high resale value
        return 7
    elif make_name in mid_range_brands_bc.value:   # Mid-range brands (e.g., Toyota, Honda), moderate resale value
        return 5
    elif make_name in budget_brands_bc.value:      # Budget brands (e.g., Suzuki,FIAT), lower resale value
        return 3
    else:                                          # Other brands (unknown or less recognized), lowest resale value
        return 1
```

→

Scoring Functions for Resale Value Factors

DF showing Resale Value Scores and Component Factors

→

make_name	log_mileage	age	days_in_market	mileage_score	age_score	days_in_market_score	make_name_score	resale_value_score
Hyundai	11.74	9	25	1	4	7	5	17
Ford	0.69	0	19	10	10	7	7	34
Ford	11.74	14	1	1	1	10	7	19
Cadillac	10.58	1	57	4	10	5	8	27
Chevrolet	0.69	0	192	10	10	3	7	30
Ford	11.26	4	56	1	7	5	7	20
Chevrolet	1.61	0	12	10	10	10	7	37
RAM	8.91	0	248	4	10	1	8	23
Ford	1.61	0	8	10	10	10	7	37
Chevrolet	11.18	4	37	1	7	5	7	20
Nissan	8.91	0	128	4	10	3	5	22
Honda	12.09	8	17	1	4	7	5	17
Chevrolet	8.91	0	6	4	10	10	7	31
Mazda	2.3	0	8	7	10	10	5	32
Ford	7.81	1	58	7	10	5	7	29
Jeep	11.19	3	40	1	7	5	7	20
Dodge	8.91	0	288	4	10	1	5	20
Ford	2.4	0	250	7	10	1	7	25
Toyota	2.2	0	91	10	10	3	5	28
RAM	0.0	0	292	10	10	1	8	29

After Feature Engineering

Model	R2 Score	Mean Average Error (MAE)	MAE as percentage of Mean (MAPE) (avg.price of car = 29,933 $)	Root Mean Square Error (RMSE)	RMSE as percentage of Mean (avg.price of car = 29,933 $)	Total Hours Trained
Linear Regression	0.842	4131	13.80 %	7217	24.11 %	17.31 hours
Decision Tree Regressor	0.884	3161	10.5 %	6190	20.68 %	1.25 hours
Random Forest	0.864	3488	11.65 %	6707	22.40 %	~ 8.6 hours
XGBoost	0.918	3018	10.10 %	5268	17.60 %	1.25 hours
GBT Regressor	0.886	3649	12.20 %	6137	20.50 %	11.3 hours

Insights from Feature Engineering (Lessons Learnt)

Simplified results

Model	Increase in R2 Score	Decrease in MAE	Decrease in RMSE
Linear Regression	~ 0.07	153	1953
Decision Tree Regressor	~ 0.06	104	1848
Random Forest	~ 0.08	303	2457
XGBoost	~ 0.07	48	2247
GBT Regressor	~ 0.05	-71	1806

Note:

The difference table above reflects the improvements achieved through Feature engineering applied on top of baseline results after EDA.

1.) Improved Model Performance:

The introduction of interaction terms and domain-specific features led to notable improvements across all models. R^2 scores increased by 5-8% for most models and RMSE decreased significantly, indicating that the models better captured relationships in the data.

2.) RMSE reduced significantly (in thousands) while MAE showed a smaller reduction (in hundreds):

This is because RMSE penalizes larger errors more heavily, and the Feature Engineering helped reduce these large errors by capturing high-impact relationships more effectively. MAE capped and didn't reduce much.

3.) Impact of Domain-Specific Features: :

Features like resale_value_score, maintenance_cost, and luxury_score introduced structured knowledge about vehicle-specific factors, likely reducing the model's tendency to produce large errors for certain cases.

4.) Interaction Terms and Stronger Feature Relationships:

Interaction terms (e.g., horsepower × engine_displacement) enabled the model to capture complex relationships that basic features alone may have missed.

Top Features

GBT on Feature-Engineered Dataset.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
# Feature Importance
feature_importance = model.featureImportances
features_list = pipeline_model.stages[-2].getInputCols()  # Get

# Sort features by their importance in descending order
sorted_features = [feature for feature, importance in sorted(zi

# Print ranked features from highest to lowest
print("Top 10 Features Ranked by Importance (Highest to Lowest)
for rank, feature in enumerate(sorted_features[:10], 1):  # Lim
    print(f"{rank}. {feature}")
```

```
Top 10 Features Ranked by Importance (Highest to Lowest):
1. maintenance_cost
2. log_mileage
3. horsepower
4. engine_displacement
5. torque
6. hp_x_engine_disp
7. maximum_seating
8. major_options_count
9. fuel_tank_volume
10. city_fuel_economy
```

XGB on Feature-Engineered Dataset.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
# Get feature importances from the loaded native XGBoost model
importance_dict = native_model.get_score(importance_type='weight')  #

# Assuming you have a list of feature names from the VectorAssembler
features_list = pipeline_model.stages[-2].getInputCols()  # Get the i

# Map the feature indices (f0, f1, ...) to the actual feature names s
sorted_importance = [
    (features_list[int(f[1:])], importance)
    for f, importance in importance_dict.items()
    if int(f[1:]) < len(features_list)  # Ensure the index is within
]

# Sort by importance
sorted_importance = sorted(sorted_importance, key=lambda x: x[1], rev

# Print the top 10 features with their actual names (sorted by import
print("Top 10 Features Ranked by Importance (Highest to Lowest)")
for rank, (feature, importance) in enumerate(sorted_importance[:10],
    print(f"{rank}. {feature}")
```

```
Top 10 Features Ranked by Importance (Highest to Lowest)
1. log_mileage
2. days_in_market
3. maintenance_cost
4. city_fuel_economy
5. major_options_count
6. latitude
7. manufactured_year
8. luxury_score
9. seller_rating
10. longitude
```

Decision Tree Regressor on Feature-Engineered Dataset.ipynb

File Edit View Insert Runtime Tools Help Last saved at 11:21

+ Code + Text

```
[ ]  # Feature Importance (for Decision Trees)
feature_importance = model.featureImportances
features_list = pipeline_model.stages[-2].getInputCols()  # Get featu

# Sort features by their importance in descending order
sorted_features = [feature for feature, importance in sorted(zip(feat

# Print ranked features from highest to lowest
print("Top 10 Features Ranked by Importance (Highest to Lowest):")
for rank, feature in enumerate(sorted_features[:10], 1):  # Limiting
    print(f"{rank}. {feature}")
```

```
Top 10 Features Ranked by Importance (Highest to Lowest):
1. maintenance_cost
2. log_mileage
3. horsepower
4. maximum_seating
5. torque
6. sp_name_encoded
7. hp_x_torque
8. fuel_tank_volume
9. width
10. is_new
```

Random Forest on Feature-Engineered Dataset.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
# Feature Importance
feature_importance = model.featureImportances
features_list = pipeline_model.stages[-2].getInputCols()  # Get

# Sort features by their importance in descending order
sorted_features = [feature for feature, importance in sorted(zi

# Print ranked features from highest to lowest
print("Top 10 Features Ranked by Importance (Highest to Lowest)
for rank, feature in enumerate(sorted_features[:10], 1):  # Lim
    print(f"{rank}. {feature}")
```

```
Top 10 Features Ranked by Importance (Highest to Lowest):
1. maintenance_cost
2. log_mileage
3. age
4. manufactured_year
5. horsepower
6. torque
7. luxury_score
8. resale_value_score
9. width
10. hp_x_torque
```

I am highlighting the newly feature engineered columns, which are part of top 10 features.

This shows, the high impact of these columns on the model.

Linear Regression on Feature-Engineered Dataset.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[ ]  features_list = pipeline_model.stages[-2].getInputCols()  # Get feat
coefficients = model.coefficients  # Get the coefficients from the l

# Sort features by the absolute value of their coefficients in desce
sorted_features = [feature for feature, coefficient in sorted(zip(fe

# Print the ranked features from highest to lowest
print("Top 10 Features Ranked by Coefficient Magnitude (Highest to L
for rank, feature in enumerate(sorted_features[:10], 1):  # Limiting
    print(f"{rank}. {feature}")
```

```
Top 10 Features Ranked by Coefficient Magnitude (Highest to Lowest):
1. horsepower
2. maintenance_cost
3. engine_displacement
4. is_new
5. hp_x_torque
6. luxury_score
7. resale_value_score
8. torque
9. width
10. wheelbase
```

Bagging

I implemented **Bagging** (Bootstrap Aggregating) as an ensemble method on both XGBoost and Decision Trees.

By training multiple instances of these models on random subsets of data and averaging their predictions, I aim to improve model stability, reduce variance, and increase predictive accuracy. This approach, sometimes referred to as Randomized XGBoost or Bootstrap Aggregating with Decision Trees, leverages the strengths of each model in a bagging ensemble for robust results.

Results

XGB

Model	R2 Score	MAE	MAPE	RMSE	RMSE as % of Mean
Baseline (after Feature Engineering)	0.918	3018	10.08 %	5268	17.60 %
Bagging	0.931	**1290**	**4.30 %**	**1820**	**6.08 %**

GBT

Model	R2 Score	MAE	MAPE	RMSE	RMSE as % of Mean
Baseline (after Feature Engineering)	0.884	3161	10.50 %	6190	20.68 %
Bagging	0.895	**1356**	**4.53 %**	**2293**	**7.66 %**

Note : Here the Baseline is the results of Feature Engineering.

Insights from Bagging (Lessons Learnt)

Improved R^2 Score:

Bagging significantly increased the R^2 scores for both XGBoost and GBT, indicating better predictive accuracy.

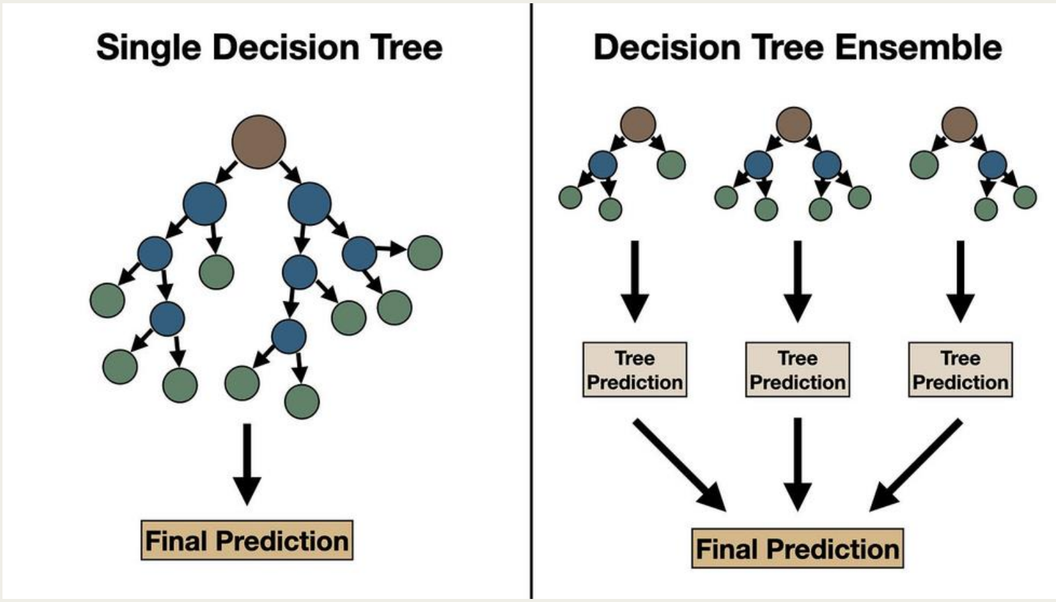
For XGBoost, R^2 improved from 0.918 to 0.930, and for GBT, it went from 0.884 to 0.895.

Highest Reduction in MAE and RMSE:

The Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) dropped drastically with bagging. For XGBoost, RMSE reduced from 5268 to 1820, and MAE decreased from 3018 to 1290. Similarly, for GBT, RMSE decreased from 6190 to 2293, and MAE reduced from 3161 to 1356. This indicates that bagging helps reduce large deviations between predicted and actual values.

Enhanced Model Stability:

Bagging enhanced model stability by training multiple models on random data subsets and averaging their predictions. This process reduced variance and minimized overfitting since each model learned different patterns from its subset. The averaging of predictions smoothed out individual model errors, leading to more reliable and consistent results. Additionally, bagging helped capture broader data patterns, allowing the ensemble to generalize better to unseen data. Overall, bagging significantly reduced error metrics and provided more robust performance compared to a single model approach.



Experiments

After Hyper parameter tuning

Model	R2 Score	Mean Average Error (MAE)	MAE as percentage of Mean (MAPE) (avg.price of car = 29,933 $)	RMSE	RMSE as percentage of Mean (avg.price of car = 29,933 $)
Linear Regression	0.844	4097	13.70 %	7204	24.06 %
Decision Tree Regressor	0.889	3232	~ 10.80 %	6577	21.97 %
Random Forest	0.879	3442	11.50 %	6543	21.85 %
XGBoost	0.928	2759	9.40 %	4964	16.60 %
GBT Regressor	0.888	3717	12.40 %	6071	20.30 %

Comparison before and after training with Best Hyper Parameters

Decision Trees

Before

Old parameters: `{'maxDepth': 15, 'maxBins': 128, 'minInstancesPerNode': 5,'minInfoGain': 0.01}`

R-Squared Score (Accuracy): ***88.38 %***

Additional Metrics:

Mean Absolute Error: 3161

Root Mean Squared Error: 7204

After

Best parameters: `{'maxDepth': 15, 'maxBins': 128, 'minInstancesPerNode': 10,'minInfoGain': 0.01}`

R-Squared Score (Accuracy): ***88.90 %***

Additional Metrics:

Mean Absolute Error: 3232

Root Mean Squared Error: 6577

GBT

Before

Old parameters: `{'maxDepth': 5, 'maxIter': 100}`

R-Squared Score (Accuracy): ***88.57 %***

Additional Metrics:

Mean Absolute Error: 3649

Root Mean Squared Error: 6137

After

Best parameters: `{'maxDepth': 5,'maxIter': 100,'stepSize': 0.1,'minInstancesPerNode': 10,'maxBins': 50}`

R-Squared Score (Accuracy): ***88.82 %***

Additional Metrics:

Mean Absolute Error: 3717

Root Mean Squared Error: 6071

XGB

Before

Old parameters : `{'maxDepth': 6, 'numRound': 100}`

R-Squared Score (Accuracy): ***91.84 %***

Additional Metrics:

Mean Absolute Error: 3018

Root Mean Squared Error: 5268

After

Best parameters : `{'maxDepth': 8, 'numRound': 200, 'eta': 0.1}`

R-Squared Score (Accuracy): ***92.75 %***

Additional Metrics:

Mean Absolute Error: 2759

Root Mean Squared Error: 4964

Linear Regression

Before

Old parameters:

`{**No specific parameters** : lr = LinearRegression(featuresCol="scaled_features", labelCol="price")}`

R-Squared Score (Accuracy): ***84.20 %***

Additional Metrics:

Mean Absolute Error: 4131

Root Mean Squared Error: 7217

After

Best parameters: `{'regParam': 0.5, 'elasticNetParam': 1.0, 'maxIter': 100}`

R-Squared Score (Accuracy): ***84.41 %***

Additional Metrics:

Mean Absolute Error: 4097

Root Mean Squared Error: 7204

Random Forest

Before

Old parameters: `{'numTrees': 50, 'maxDepth': 10, 'minInstancesPerNode': 10}`

R-Squared Score (Accuracy): ***86.35 %***

Additional Metrics:

Mean Absolute Error: 3488

Root Mean Squared Error: 6707

After

Best parameters: `{'numTrees': 100, 'maxDepth': 20, 'minInstancesPerNode': 10}`

R-Squared Score (Accuracy): ***87.92%***

Additional Metrics:

Mean Absolute Error: 3442

Root Mean Squared Error: 6543

Insights from Hyper parameter tuning (Lessons Learnt)

Simplified results

Model	Increase in R2 Score	Decrease in MAE	Decrease in RMSE
Linear Regression	0.0021	34	13
Decision Tree Regressor	0.0052	-71	-387
Random Forest	0.0157	46	164
XGBoost	0.0091	259	304
GBT Regressor	0.0025	-68	66

Note:

The difference table above reflects the improvements achieved through hyperparameter tuning (HPT) applied on top of the feature-engineered baseline.

Improved Model Performance:

Hyperparameter tuning led to small yet meaningful increases in R^2 scores across all models, with the highest increased in Random forest followed by XGB. These improvements indicate that tuning helped the models explain a slightly higher variance in the target variable, enhancing overall accuracy.

Reduction in Error Metrics (MAE and RMSE):

XGBoost experienced the most significant reduction in both MAE demonstrating the effectiveness of hyperparameter tuning in refining prediction precision. Similarly, Random Forest and Linear Regression saw moderate reductions in RMSE and MAE, indicating improved prediction stability and reduced error.

Enhanced Stability Across Models:

Although some models like Decision Tree and GBT Regressor showed minor increases in MAE and RMSE, most models displayed stable or improved performance. The overall reduction in error metrics suggests that tuning helped the models generalize better, balancing the trade-off between variance and bias effectively.

Sub - Sampling

Row - wise sampling

Size of DF (Training +Testing)	R2 Score	Mean Average Error (MAE)	Root Mean Square Error (RMSE)
100k	0.785	3341	8774
200k	0.860	3237	6975
300k	0.874	3746	6674
600k	0.884	3161	6190
1 million	0.865	3144	6794

Decision Trees

Size of DF (Training +Testing)	R2 Score	Mean Average Error (MAE)	Root Mean Square Error (RMSE)
100k	0.904	3009	5971
200k	0.907	2938	5509
300k	0.918	3018	5268
600k	0.921	2975	6581

XGB

Row - wise sampling

Size of DF (Training +Testing)	R2 Score	Mean Average Error (MAE)	Root Mean Square Error (RMSE)
100k	0.753	3517	9420
200k	0.830	3545	7687
300k	0.846	3594	6981
600k	0.864	3488	6707

Random Forest

Size of DF (Training +Testing)	R2 Score	Mean Average Error (MAE)	Root Mean Square Error (RMSE)
100k	0.794	3604	8767
200k	0.856	3638	7073
300k	0.872	3654	6549
600k	0.886	3649	6137

GBT

Size of DF (Training +Testing)	R2 Score	Mean Average Error (MAE)	Root Mean Square Error (RMSE)
100k	0.795	4470	8573
200k	0.816	4318	7153
300k	0.836	3916	7457
600k	0.842	4131	7217

Linear Regression

Column - wise sampling

Note:
For XGB and Decision Trees, I ran 10 iterations with 70% randomly selected features on each iteration.

Now, I am displaying the best and worst results from the different iterations I ran.

Baseline Results

Model Name	R2 Score	Mean Average Error (MAE)	Root Mean Square Error (RMSE)
XGB	0.918	3018	5268
Decision Trees	0.884	3161	6190

Worst case

Model Name	R2 Score	Mean Average Error (MAE)	Root Mean Square Error (RMSE)
XGB	0.894	3300	5990
Decision Trees	0.85	3335	7118

Best case

Model Name	R2 Score	Mean Average Error (MAE)	Root Mean Square Error (RMSE)
XGB	0.921	2959	51845
Decision Trees	0.887	3135	6104

Insights from Sub-sampling (Lessons Learnt)

Row wise Sub Sampling:

1. Improved Performance with Larger Data:

All models generally show an improvement in R^2 scores and a reduction in MAE and RMSE as the data size increases.

This suggests that larger datasets help capture underlying patterns more effectively, leading to better model accuracy and reduced error rates.

2. Diminishing Returns at Higher Data Sizes:

While performance improves with increased data size, the gains start to plateau, particularly between 300k and 600k rows.

This indicates that after a certain point, additional data yields only marginal improvements, suggesting that the models may have captured most of the essential patterns by this stage.

Column Sub Sampling:

1. Iteration Results:

Iterative training with 70% of features demonstrated consistently high accuracy (R^2 between 0.894and 0.921 for XGB), indicating that the model performs reliably across different feature subsets.

2. Overall Performance:

The overall performance of the model was strong, with consistent accuracy and error metrics across multiple iterations. This indicates stability in model performance, regardless of feature selection variability. Such stability suggests that the model can generalize well to unseen data.

3. Feature Redundancy:

The 1-2% difference in performance when using 70% of columns compared to 100% suggests that many of my features likely contain redundant or correlated information. In other words, some features might be providing similar information, which means that I can achieve comparable predictive performance without needing all the original columns.

BERT Embeddings for Description Column: Feature Transformation

In this method, I used BERT embeddings to transform the text data into meaningful numerical representations, capturing semantic details of the descriptions. I then incorporated these embeddings into the existing DF for model training, aiming to evaluate the added impact of BERT on predictive performance. This allowed me to compare the model's results with and without the enriched description feature.

Steps:

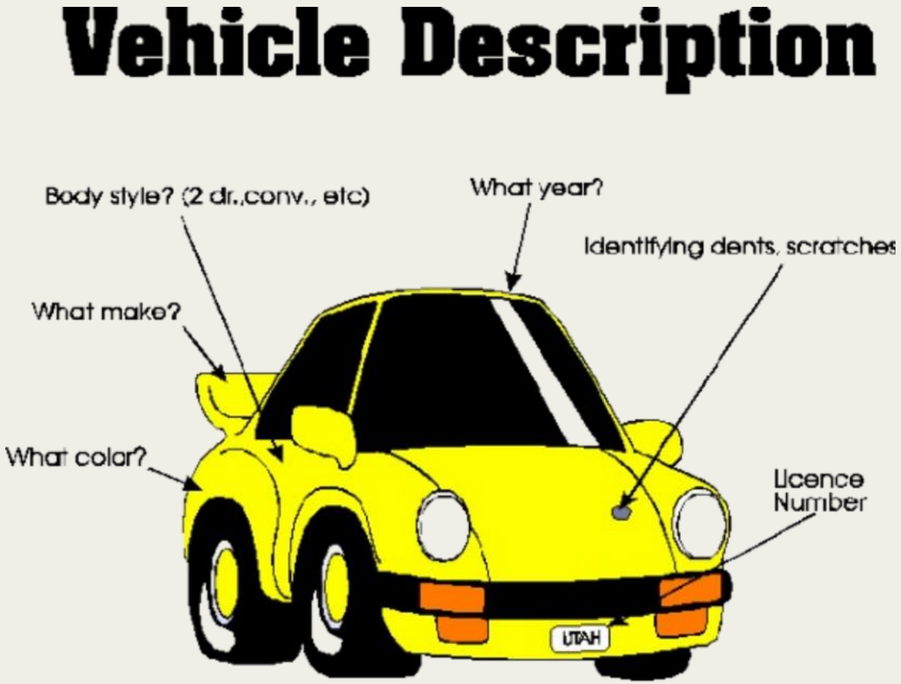
1. Loaded pre-trained BERT tokenizer and model from Hugging Face for embedding generation.

2. Defined a function (get_bert_embeddings) to tokenize text, feed it to BERT, and extract the CLS token as the sentence representation.

3. Wrapped the function as a PySpark UDF and applied it to the 'description' column in the Spark DataFrame.

4. Added resulting embeddings as a new column `bert_embeddings` in the DataFrame (df_with_bert).

5. Trained Decision Trees and XGBoost models on `df_with_bert`, with and without the 'description' column, to study the impact of transformed descriptions on model performance.



Insights from Transformation of 'Description' (Lessons Learnt)

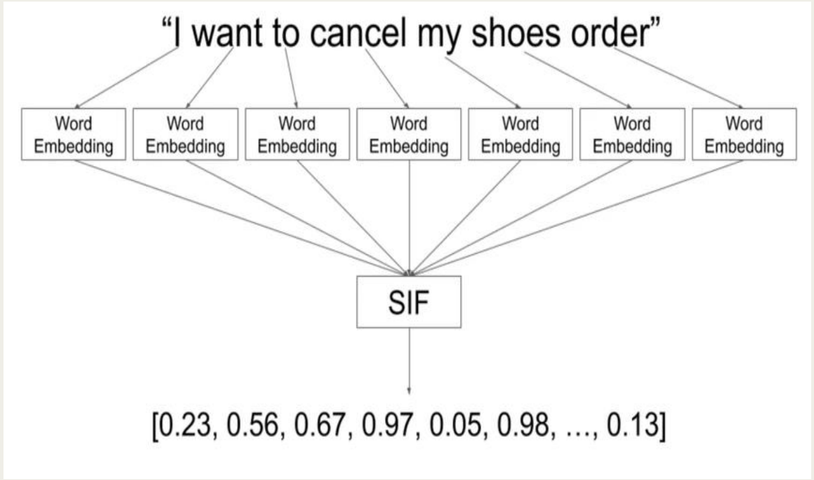
Results

Model Name	Dataset Size	R2 Score	MAE	RMSE
XGBoost (without BERT)	100k	0.86	2994	6955
XGBoost **(with BERT)**	100k	0.87	3007	6892

Model Name	Dataset Size	R2 Score	MAE	RMSE
GBTRegressor (without BERT)	100k	0.79	3471	8556
GBTRegressor **(with BERT)**	100k	0.84	3948	7257

Inferences

1. BERT embeddings led to inconsistent model improvements.
2. GBTRegressor saw a 4% accuracy boost with BERT, but training/testing time increased to 9 hours (9 X increase).
3. XGBoost showed only a 1% accuracy gain with BERT embeddings.
4. RMSE and MAE did not decrease significantly, suggesting limited performance benefits from BERT embeddings despite the usage of increased resources.



log_mileage	major_options_count	bert_embeddings
11.29	0	[-0.3094957172870636, -0.03846015781164169, 0....
8.91	0	[-0.44042280316352844, 0.12084095180034637, -0...
11.18	1	[-0.5281291604042053, 0.03638622537255287, 0.3

Final Comparison

Baseline

R2 Score: 0.854
MAE: 3066
MAPE: 10.24 %
RMSE: 7515
RMSE as percentage of Mean: 25.10 %

Final Result

R2 Score: 0.931
MAE: 1290
MAPE: 4.30 %
RMSE: 1820
RMSE as percentage of Mean: 6.08 %

Note:
I chose XGBoost as the comparison model because it produced the best results after applying bagging.
Therefore, I'm using the XGBoost baseline as the reference point for measuring improvement.

Best Results from:

The approaches that gave the best result was
1. Bagging (on XGB and Decision Trees)
2. Feature Engineering (after Bagging, this showed the most improvement)

Both these approaches provided higher improvements in R^2, MAE (MAPE), and RMSE.

Summary of Lessons Learnt

General Inferences:

a. Feature Engineering **Greatly decreased** RMSE and substantially boosted the R^2 score by capturing complex relationships through interaction terms and domain-specific features. (RMSE decreased by approximately 2000 on average, and R^2 increased by 5-8% for each model.)

b. Subsampling had a minimal impact on performance, with only slight improvements in model accuracy.

c. Bagging approach with XGB delivered the **Best results**, enhancing model stability and predictive power by combining multiple model outputs.

MAPE after XGB-Bagging model was ~ 4%, meaning that, on average, the model's predictions deviated by 4% from the actual values.

a. Hyperparameter Tuning (HPT) yielded a moderate improvement, with a maximum increase of 1.5% in R^2, further refining the model's performance on top of the feature-engineered baseline.

Model Results:

1. **Tree-Based Models** (Decision Tree, Random Forest, GBT Regressor)
   demonstrated strong performance with all the R^2 scores above 87% in minimum, and moderate error metrics, effectively capturing non-linear relationships and providing reliable predictions for complex datasets.

2. **Linear Regression**
   Underperformed compared to tree-based and boosting models, with the lowest R^2 score (84.41%) and the highest RMSE. This suggests its limitations in handling non-linear relationships and complex patterns present in the data.

3. **Boosting Model** (XGBoost)
   Outperformed all other models, achieving the highest R^2 score (93.10%) and the lowest error metrics. This model's ability to refine errors iteratively enabled it to capture intricate patterns more accurately, making it the best choice for this prediction task.

Analysis of Most accurate result

On the results of Bagging with XGB, I identified predictions with an error percentage between **0% and 1%**, joined them with the original dataset to include all features, and selected relevant columns (price, final prediction, error percentage, and all original features) for further analysis.

1. Filtered the Best Performing Data:

I created a subset of the data by selecting rows where the model's prediction error was less than 1%, representing cases with high prediction accuracy. I then joined this subset with the original dataset to retain all relevant features for further analysis.

2. Calculated Statistics for Numeric Scores:

I calculated statistical metrics (mean, standard deviation, min, max, and mode) for each of the engineered numeric score columns, allowing me to understand the distribution of these features in the best-performing subset.

3. Examined Top Values for Categorical Columns:

I identified the categorical columns in the dataset and displayed the top three most frequent values for each. This helped me understand the common categories for features like fuel type, body type, transmission, etc., among instances where the model performed well.

The following factors were associated with the least prediction errors and highest model accuracy: (<=1% error)

1. Low mileage, moderate age, and newer model years (especially 2015-2020).
2. Popular body types (SUVs and sedans) and mid-size seating capacities.
3. Mainstream brands (Ford, Chevrolet, Nissan) and common engine types (I4, V6, V8).
4. Gasoline and hybrid fuel types, along with popular colors (white, black, silver).
5. Well-equipped vehicles with advanced features and premium options.

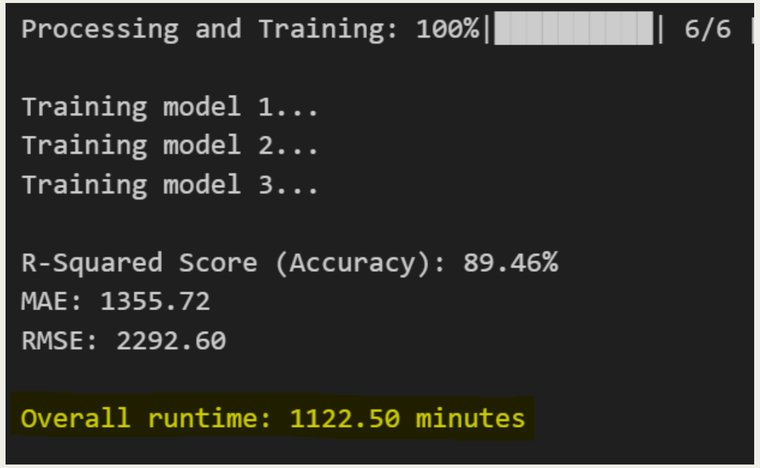
Challenges

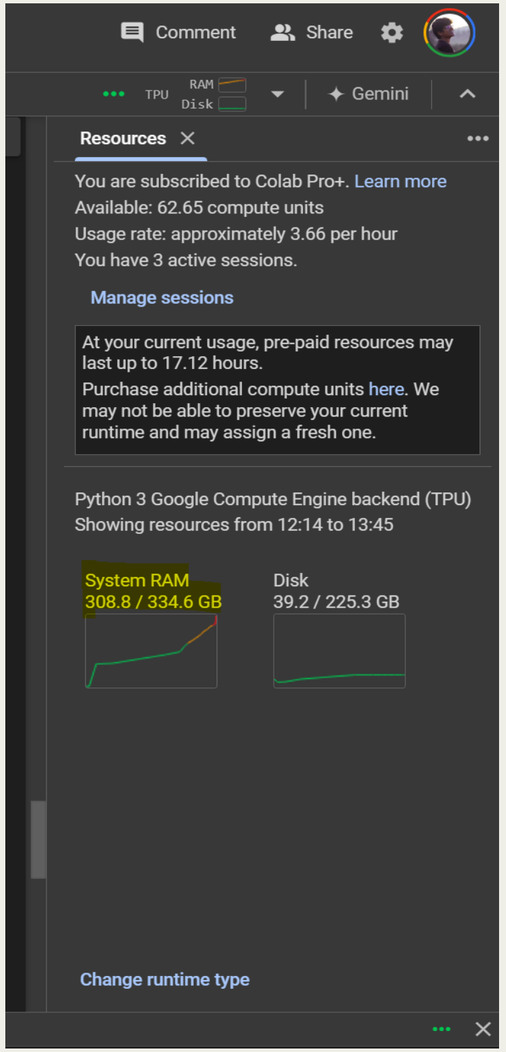
1. **High Memory Usage:**
   Training complex models required substantial RAM, with memory usage reaching up to 308 GB for certain configurations.

2. **Very Long Training Times:**
   Some models, like Linear Regression with the best parameters, took up to 23 hours to train. Random Forest and GBT also took more than 12 hours on average after HPT. Bagging with Decision trees (shown in the image), also took nearly 19 hours to complete as the task was complex.

3. **Learning Domain Knowledge:**
   Developing additional features like the resale score required a deep understanding of the car market and industry-specific terminology. I spent significant time learning these aspects, which added complexity to the feature engineering process.

4. **Significant Compute Power Needed:**
   The combination of high RAM usage and long training times highlights the need for robust computational infrastructure to handle the workload effectively.





Thank you!