Regression-Based Predictive Modelling of Car Selling Prices

-By Saloni kamboj & Apeksha Nanda

Table of Contents:

1. Introduction

This assignment's main objective is to create a reliable predictive model that can be used to estimate car costs based on a number of features. The problem is approached as a regression task where the target variable is the selling price of the cars. In order to evaluate and discover patterns from a dataset including details on the qualities of automobiles, this project makes use of a variety of machine learning algorithms.

The implementation constraints include issues related to memory and time utilisation. Memory and time profiling were used in the analysis to evaluate the models' effectiveness in terms of computational needs and prediction accuracy. These constraints help us in creating a balance between efficiency and accuracy By addressing these constraints, the project aims to create a solution that is not only computationally feasible but also capable of delivering precise and reliable predictions.

The design decisions are also impacted by broader societal, economic, and ethical aspects in addition to the technical aspects. Economic factors come into play as the automotive market dynamics impact the dataset and subsequently influence the design choices. There are also societal factors of our dataset needing to be fair, unbiased and ethical. The project's design philosophy is based on striking this balance between technical efficiency, economic relevance, and ethical integrity.

The dataset is composed of 8,128 samples, which has several features such as year, mileage, engine specs, maximum power, seats, and more. The dataset is the basis for training and assessing machine learning models. It can be obtained via Kaggle at [https://www.kaggle.com/datasets/nehalbirla/vehicle-dataset-from-cardekho?select=Car+details+v3.csv]. This large dataset offers a complete examination and analysis of the factors impacting vehicle pricing in the market, in addition to strengthening the predictive model.

2. Data exploration and preparations

In the first milestone of our research project, we used machine learning (ML) approaches to predict car selling prices using regression analysis. To understand the data, the first steps were to load the dataset, import the required libraries, and examine its structure. To address missing values, data preparation was done. Descriptive statistics were produced for every column, offering insights into the features of the dataset.

The dataset has to be cleaned before it can be used for training. Rows containing null values were deleted to remedy missing data. To improve their modelling applicability, numerical characteristics including mileage, engine, and max power were also converted from string to float types. After a careful check for duplicate values in the dataset, no duplicates were found. Our data integrity is guaranteed by this cleaning procedure, which also improves the precision of our prediction models.

```
df.isna().sum()

/usr/local/lib/python3.10/di
  return method()
\begin{tabular}{lr}
\toprule
{} &    0 \\
\midrule
name          &    0 \\
year          &    0 \\
selling\_price &    0 \\
km\_driven     &    0 \\
fuel          &    0 \\
seller\_type   &    0 \\
transmission  &    0 \\
owner         &    0 \\
mileage       &  221 \\
engine        &  221 \\
max\_power     &  215 \\
torque        &  222 \\
seats         &  221 \\
\bottomrule
\end{tabular}
```

```
[ ] df.isnull().sum()

/usr/local/lib/python3.10/
  return method()
\begin{tabular}{lr}
\toprule
{} &    0 \\
\midrule
name          &    0 \\
year          &    0 \\
selling\_price &    0 \\
km\_driven     &    0 \\
fuel          &    0 \\
seller\_type   &    0 \\
transmission  &    0 \\
owner         &    0 \\
mileage       &  221 \\
engine        &  221 \\
max\_power     &  215 \\
torque        &  222 \\
seats         &  221 \\
\bottomrule
\end{tabular}
```

```
[ ] df.dropna(inplace=True)

df.isnull().sum()

/usr/local/lib/python3.10/d
  return method()
\begin{tabular}{lr}
\toprule
{} &  0 \\
\midrule
name          &  0 \\
year          &  0 \\
selling\_price &  0 \\
km\_driven     &  0 \\
fuel          &  0 \\
seller\_type   &  0 \\
transmission  &  0 \\
owner         &  0 \\
mileage       &  0 \\
engine        &  0 \\
max\_power     &  0 \\
torque        &  0 \\
seats         &  0 \\
\bottomrule
\end{tabular}
```

```
df.describe()

'content/drive/MyDrive/Colab Notebooks/mlwpy_code/mlwpy.py:55: FutureWarning: In future versions `DataFrame.to_
  return "{\centering\n%s\n\medskip}" % self.to_latex()
```

	year	selling_price	km_driven	seats
count	7,906.0000	7,906.0000	7,906.0000	7,906.0000
mean	2,013.9839	649,813.7208	69,188.6598	5.4164
std	3.8637	813,582.7484	56,792.2963	0.9592
min	1,994.0000	29,999.0000	1.0000	2.0000
25%	2,012.0000	270,000.0000	35,000.0000	5.0000
50%	2,015.0000	450,000.0000	60,000.0000	5.0000
75%	2,017.0000	690,000.0000	95,425.0000	5.0000
max	2,020.0000	10,000,000.0000	2,360,457.0000	14.0000

Data visualisations are essential for understanding the dataset and directing the creation of models. Scatter plots are among the visualisations that highlight significant facets of the data distribution. Various Graphs are shown throughout the project to explain the linearity of the predictive models. These will be shown in later parts of the report

3. Baseline methods

We looked at two methods for training and dividing our data: building a custom function for data division and using the scikit-learn module. In the end, we decided to use the train_test_split module from scikit-learn for the first approach. Our dataset may be easily divided into training and testing sets with the help of this common module. In particular, we ensured that each split received 25% of the data by setting the test_size argument to 25%. Although creating a new function is a possible alternative, scikit-learn's train_test_split is preferred because of its extensive usage, reliable functioning, and simplicity of customization. We used K-Nearest Neighbours (KNN) regression models and linear regressor to improve the precision of our predictions. The linear regression model was assessed using metrics like Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared on a validation set. The model was trained on parameters including year, km travelled, mileage, engine, max power, and seats.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

features = df[['year', 'km_driven', 'mileage', 'engine', 'max_power', 'seats']]
target = df['selling_price']

#split the data
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.25, random_state=42)

#create validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, random_state=42)

# Create a linear regression model
linear_model = make_pipeline(StandardScaler(), LinearRegression())

# Train the model on the training set
linear_model.fit(X_train, y_train)

# Make predictions on the validation set
linear_predictions = linear_model.predict(X_val)
from sklearn.neighbors import KNeighborsRegressor
import numpy as np

# Create a KNN regression model
knn_model = make_pipeline(StandardScaler(), KNeighborsRegressor(n_neighbors=3))  # You can adjust the number of
neighbors as needed

# Train the KNN model
knn_model.fit(X_train, y_train)

# Make predictions on the validation set
knn_predictions = knn_model.predict(X_val)
# Evaluate the model on the validation set
val_mae = mean_absolute_error(y_val, linear_predictions)
val_mse = mean_squared_error(y_val, linear_predictions)
val_rmse = np.sqrt(mean_squared_error(y_val, linear_predictions))
val_r2 = r2_score(y_val, linear_predictions)
```

```
print(f'Mean Absolute Error: {val_mae}')
print(f'Mean Squared Error: {val_mse}')
print(f'Root Mean Squared Error: {val_rmse}')
print(f'R-squared: {val_r2}')
knn_val_mae = mean_absolute_error(y_val, knn_predictions)
knn_val_mse = mean_squared_error(y_val, knn_predictions)
knn_val_rmse = np.sqrt(mean_squared_error(y_val, knn_predictions))
knn_val_r2 = r2_score(y_val, knn_predictions)


print(f'Mean Absolute Error: {knn_val_mae}')
print(f'Mean Squared Error: {knn_val_mse}')
print(f'Root Mean Squared Error: {knn_val_rmse}')
print(f'R-squared: {knn_val_r2}')
```

Upon evaluating the baseline models, the analysis based on the R-Squared error metric revealed that KNN outperformed Linear Regressor by a significant margin of at least 32%. KNN showed lower Mean Absolute Error and Mean Squared Error values. Even with these improved performance measures, it's important to take into account other useful aspects like memory usage and time utilisation. The linear regression and KNN models were subjected to memory and time profiling to assess how much memory they used and how efficiently they computed predictions on the larger validation set. Execution time was monitored by %timeit, and memory use was disclosed by the %memit command. These evaluations are essential to the selection of the model since they direct our choice based on prediction accuracy and resource efficiency. The resulted output is
peak memory for linear model: 485.42 MiB, increment: 0.00 MiB
peak memory for KNN : 485.42 MiB, increment: 0.00 MiB
run time for linear model: 1.62 ms ± 237 µs per loop
run time for KNN : 16 ms ± 748 µs per loop
Notably, as compared to KNN, Linear Regression showed an edge in terms of quicker prediction times. Finding the best model for the unique restrictions and implementation requirements requires weighing the trade-off between computational efficiency and forecast accuracy.

Metrics/ Resource utilisation	Linear Regressor	KNN
Mean Absolute Error (MAE)	288105.70962602773	93356.43650258484
Mean Squared Error (MSE)	229931603624.36816	39073007309.15697
R-squared (R^2)	0.6548100816089562	0.9413407813813415
Memory Usage	485.42 MiB	485.42 MiB
Time Utilisation	1.62 ms ± 237 µs per loop	16 ms ± 748 µs per loop

The nonlinearity observed in the scatter plot suggests that a linear regression model might oversimplify the true underlying patterns. On the other hand, the linearity observed in the scatter plot for K-Nearest Neighbors (KNN) indicates that this algorithm successfully captures local linear relationships within the data.

4. Advance methods

In exploring advanced models for our predictive task, we carefully considered a range of options and alternatives. Among the models evaluated were Linear Discriminant Analysis, Support Vector Machine (SVM), Random Forest, and Gradient Boosting Regression. After a thorough analysis, we chose Random Forest and Gradient Boosting as they demonstrate superior capabilities in handling non-linear relationships within the data. These models offer flexibility and are known for their efficiency in handling complex patterns, making them optimal choices for our regression problem. The decision to opt for these models was grounded in their ability to provide accurate predictions while accommodating intricate relationships present in the dataset.

To experiment with the selected options, we implemented Random Forest and Gradient Boosting models, leveraging their inherent strengths in capturing non-linear patterns. The choice to pursue these models was reinforced by their superior performance metrics, aligning with our objective of achieving accurate predictions. Throughout the experimentation process, we incorporated creative elements such as feature engineering, hyperparameter tuning, and extensive data visualisation on the existing Random Forest model. These endeavours aimed to enhance the models' predictive capabilities and uncover latent patterns within the dataset.

Evaluation of these advanced models was conducted using a comprehensive set of performance metrics, including Mean Absolute Error, Mean Squared Error, and R2 Score. These metrics provided a holistic assessment of predictive accuracy and model performance using validation sets. Additionally, memory profiling was employed to scrutinise the resource utilisation and time efficiency of the models. The results indicated that both Random Forest and Gradient Boosting performed admirably, showcasing significantly lower errors compared to simpler models like Linear Regression or KNN. Furthermore, both models exhibited high R2 scores, indicating robust predictive capabilities. While Random Forest demonstrated a slight advantage in accuracy, Gradient Boosting outperformed in terms of resource utilisation, demonstrating a nuanced trade-off between predictive performance and computational efficiency.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline


features = df[['year', 'km_driven', 'mileage', 'engine', 'max_power', 'seats']]
target = df['selling_price']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.25, random_state=42)
#split into validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, random_state=42)
```

```
# Random Forest Regressor
rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(X_train, y_train)

# Predictions
rf_predictions = rf_model.predict(X_val)

# Evaluate the Random Forest model
rf_mae = mean_absolute_error(y_val, rf_predictions)
rf_mse = mean_squared_error(y_val, rf_predictions)
rf_r2 = r2_score(y_val, rf_predictions)

print("Random Forest Regressor Metrics:")
print(f'Mean Absolute Error: {rf_mae}')
print(f'Mean Squared Error: {rf_mse}')
print(f'R-squared: {rf_r2}')

# Gradient Boosting Regressor
gb_model = GradientBoostingRegressor(random_state=42)
gb_model.fit(X_train, y_train)

# Predictions
gb_predictions = gb_model.predict(X_val)

# Evaluate the Gradient Boosting model
gb_mae = mean_absolute_error(y_val, gb_predictions)
gb_mse = mean_squared_error(y_val, gb_predictions)
gb_r2 = r2_score(y_val, gb_predictions)

print("\nGradient Boosting Regressor Metrics:")
print(f'Mean Absolute Error: {gb_mae}')
print(f'Mean Squared Error: {gb_mse}')
print(f'R-squared: {gb_r2}')
```

The average absolute difference (MAE) for the Random Forest Regressor between the projected and actual selling prices is around $75,127. An increased sensitivity to huge mistakes is indicated by the related MSE of about 23472129077. A strong fit is indicated by the extremely high R^2 value of around 0.9647619457564942, which indicates that the Random Forest model accounts for nearly 96.5% of the variance in selling prices.

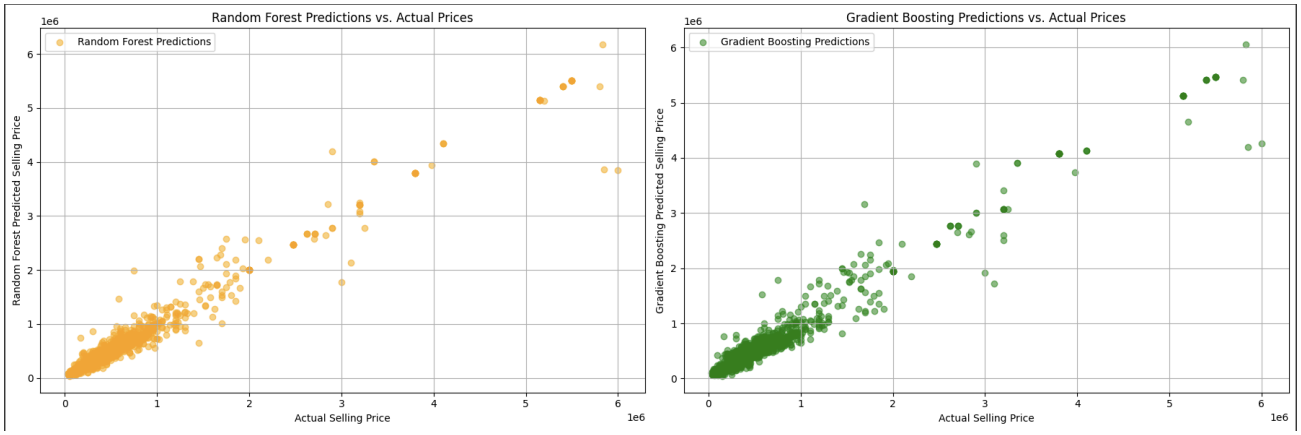
Using the Gradient Boosting Regressor, it is possible to infer a bigger average absolute difference between projected and actual prices due to the model's greater MAE of around $93,505 when compared to the Random Forest model. Additionally, the MSE is larger at around 26100710533, suggesting a greater susceptibility to huge mistakes. However, the R^2 value of around 0.9608157295599149 indicates that the gradient-boosting model accounts for nearly 96.1% of the variation in selling prices, demonstrating a high degree of predictive power.

To sum up, both models show strong predictive performance, as evidenced by their high R^2 values. In terms of MAE and MSE, the Random Forest model performs better, indicating improved accuracy and resistance to outliers. Factors like dataset features, interpretability, and computing efficiency may play a role in which model is selected.

The Random Forest and Gradient Boosting models were then subjected to memory and time profiling to assess how much memory they used and how efficiently they computed predictions on the larger validation set. Execution time was monitored by %timeit, and memory use was disclosed by the %memit command. These evaluations are essential to the selection of the model since they direct our choice based on prediction accuracy and resource efficiency. The resulted output is
peak memory for random forest model: 485.51 MiB, increment: 0.00 MiB
peak memory for gradient boosting: 485.51 MiB, increment: 0.00 MiB
run time for random forest model: 88.6 ms ± 32.9 ms per loop
run time for gradient boosting: 17.4 ms ± 5.28 ms per loop
Based on this method of resource utilisation gradient boosting turns out to be better than random forest.

Metrics/ Resource utilization	Random Forest	Gradient Boosting
Mean Absolute Error (MAE)	75127.64270478013	93505.52040584374
Mean Squared Error (MSE)	23472129077.747635	26100710533.884052
R-squared (R^2)	0.9647619457564942	0.9608157295599149
Memory Usage	485.51 MiB	485.51 MiB
Time Utilisation	88.6 ms ± 32.9 ms per loop	17.4 ms ± 5.28 ms per loop

In this analysis, the performance graphs of both Random Forest and Gradient Boosting models show a nearly linear convergence, with small differences in their errors. The visual representation of these models poses challenges in differentiation due to the similarity in their graphs. As observed in the graphs of Random Forest and Gradient Boosting, they may appear almost identical at first glance. However, Upon closer examination, The data points in the graph corresponding to Random Forest are narrower compared to those of Gradient Boosting. Despite the overall visual similarity, the narrower spread signifies that Random Forest produces lesser errors compared to Gradient Boosting.



5. Feature Engineering methods

In the feature engineering phase of the project, two distinct approaches were explored: manual feature engineering and automatic feature engineering using a pipeline. The motivation for undertaking feature engineering stemmed from the initial assessment of feature importance derived from the Random Forest model. Feature importance analysis hinted at the potential to improve model performance by modifying or introducing specific features.

For manual feature engineering, a new feature, 'max_power_year_interaction,' was created by taking the product of 'max_power' and 'year.' This interaction term was introduced based on the intuition that the combination of these two features might capture nuanced relationships not adequately represented by the original features. The model was retrained using the manually engineered features, and predictions were made to evaluate the impact on performance.

```
# Original features
features_original = df[['year', 'km_driven', 'mileage', 'engine', 'max_power', 'seats']]

# Create a new feature: Interaction between 'max_power' and 'year'
features_original['max_power_year_interaction'] = features_original['max_power'] * features_original['year']

# Display the modified features
print("Modified Features:")
print(features_original.head())

# Update the features used in modeling
me_features = features_original[['year', 'km_driven', 'mileage', 'engine', 'max_power', 'seats',
'max_power_year_interaction']]

# Retrain the model (Random Forest, for example) with the manually engineered features
X_train_manual, X_test_manual, y_train_manual, y_test_manual = train_test_split(me_features, target,
test_size=0.25, random_state=42)
X_train_manual, X_Val_manual, y_train_manual, y_val_manual = train_test_split(X_train_manual, y_train_manual,
test_size=0.25, random_state=42)


# Create and train the model
me_model = RandomForestRegressor(random_state=42)
me_model.fit(X_train_manual, y_train_manual)

# Make predictions on the test set
me_predictions = me_model.predict(X_Val_manual)
```

On the other hand, automatic feature engineering involved using a pipeline with PolynomialFeatures to create interaction terms and polynomial features. The pipeline was applied to the original features, generating a set of automatically engineered features. The Random Forest model was retrained using these features, and predictions were made to assess the model's performance.

```
features_original = df[['year', 'km_driven', 'mileage', 'engine', 'max_power', 'seats']]

# Create a pipeline for automated feature engineering
feature_engineering_pipeline = Pipeline([
    ('interaction', PolynomialFeatures(degree=2, interaction_only=True, include_bias=False)),
    ('polynomial', PolynomialFeatures(degree=2, include_bias=False)),
])
```

```
# Fit and transform the original features
features_auto_engineered = feature_engineering_pipeline.fit_transform(features_original)

# Display the automated features
print("Automated Features:")
print(features_auto_engineered)

# Retrain the model (Random Forest, for example) with the automatically engineered features
X_train_auto, X_test_auto, y_train_auto, y_test_auto = train_test_split(features_auto_engineered, target,
test_size=0.25, random_state=42)
X_train_auto, X_Val_auto, y_train_auto, y_val_auto = train_test_split(X_train_auto, y_train_auto, test_size=0.25,
random_state=42)

# Create and train the model
ae_model = RandomForestRegressor(random_state=42)
ae_model.fit(X_train_auto, y_train_auto)

# Make predictions on the test set
ae_predictions = ae_model.predict(X_Val_auto)
```

Upon evaluating both approaches, it was observed that neither manual nor automatic feature engineering significantly improved the performance of the Random Forest model. The metrics, including Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared, showed minimal variation compared to the model without feature engineering. This suggests that the initial features in the dataset were already sufficient in capturing the underlying patterns in the data, and additional engineered features did not contribute substantially to the predictive accuracy of the Random Forest model.

Model Performance Metrics:

Random Forest Regressor Metrics:
MAE: 75127.64
MSE: 23472129077.75
R-squared: 0.9648

Random Forest Regressor Metrics with Manual Feature Engineering:
MAE: 75597.96
MSE: 24089322771.59
R-squared: 0.9638

Random Forest Regressor Metrics with Automatic Feature Engineering:
MAE: 77475.87
MSE: 33277895323.04
R-squared: 0.9500

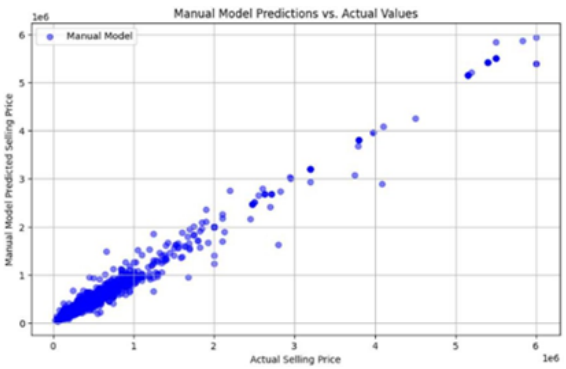
However, when considering computational cost, feature engineering proves to be useful.
memory and time profiling for manual feature engineering
peak memory: 494.06 MiB, increment: 0.00 MiB
41 ms ± 5.55 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

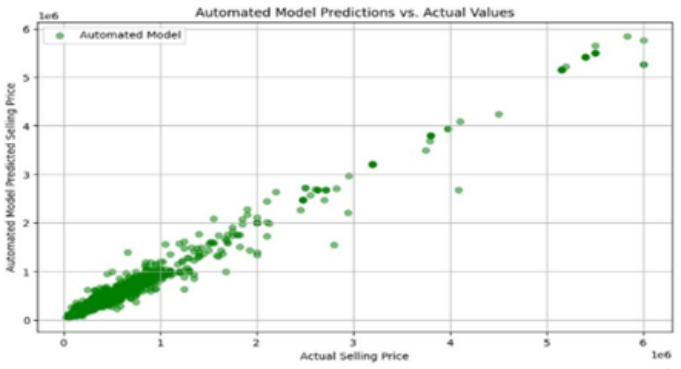
memory and time profiling for automated feature engineering
peak memory: 494.06 MiB, increment: 0.00 MiB
42.1 ms ± 556 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
memory and time profiling for random forest
peak memory: 485.51 MiB, increment: 0.00 MiB
88.6 ms ± 32.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
Both manual and automatic feature engineering added a minimal computational overhead, with peak memory remaining almost unchanged from the baseline Random Forest model. The slight increase in processing time is reasonable, given the potential benefits in predictive accuracy.

In conclusion, while feature engineering did not substantially enhance the predictive performance of the Random Forest model in this scenario, the consideration of computational cost can be important as well. Both manual and automatic feature engineering methods showed efficiency in terms of memory usage and processing time. Therefore, the decision to employ feature engineering should be made based on the dataset and thesis. In this case, SInce we are looking for more accuracy, we can say that Original random forest model is better and more useful than feature engineering models.

Metrics/ Resource utilization	Manual Feature Engineering	Automated Feature Engineering
Mean Absolute Error (MAE)	75597.95862159491	77475.87329258079
Mean Squared Error (MSE)	24089322771.591072	33277895323.0431
R-squared (R^2)	0.9638353700381022	0.9500408217499617
Memory Usage	494.06 MiB	494.06 MiB
Time Utilisation	41 ms ± 5.55 ms per loop	42.1 ms ± 556 µs per loop

Data Visualisation for Manual and automated feature engineering look similar to Random forest as there are minute differences in their predicted results.





6. Hyperparameter Tuning methods

In order to optimise the Random Forest model's performance, hyperparameter tuning was conducted using a random search approach. We researched many kinds of hyperparameter tuning methods such as manual, grid search , random search etc. We decided to perform the random search as it is faster than others (we performed grid search and compared the run times) and still produce accurate results.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

# Define the parameter distribution for Random Forest
rf_parameters = {
    'n_estimators': randint(50, 150),
    'max_depth': [None, 10, 20],
    'min_samples_split': randint(2, 10),
    'min_samples_leaf': randint(1, 4)
}

# Create the RandomizedSearchCV object
random_search_rf = RandomizedSearchCV(RandomForestRegressor(random_state=42), param_distributions=rf_parameters,
n_iter=10, cv=5, scoring='neg_mean_absolute_error', random_state=42)

# Fit the randomized search to the data
random_search_rf.fit(X_train, y_train)

# Get the best hyperparameters
best_params_rf_random = random_search_rf.best_params_
print("Best Hyperparameters for Random Forest (RandomizedSearchCV):", best_params_rf_random)

# Use the best model from the randomized search
best_rf_model= random_search_rf.best_estimator_
```

Below is the analysis of the hyperparameter tuning results and a comparison of models on the test set.

Random Forest Regressor Metrics (Before Hyperparameter Tuning):
MAE: 75127.64
MSE: 23472129077.75
R-squared: 0.9648
Tuned Random Forest Regressor Metrics (After Hyperparameter Tuning):
MAE: 71639.18
MSE: 24550121004.22
R-squared: 0.9662
The tuned Random Forest model exhibited a reduced MAE of 71639.18 compared to the original 75127.64, indicating an improvement in the model's accuracy in predicting the selling price. The MSE after hyperparameter tuning was 24550121004.22, slightly higher than the initial 23472129077.75. While MSE increased marginally, it is essential to consider that a more balanced trade-off between MAE and MSE was achieved. The R-squared value increased from 0.9648 to 0.9662 after hyperparameter tuning, indicating a better fit of the model to the variance in the target variable. The improvement, although modest, signifies a refined model.
The comparison on the test set demonstrates that the hyperparameter-tuned Random Forest model outperforms the baseline model. The reduction in MAE and the

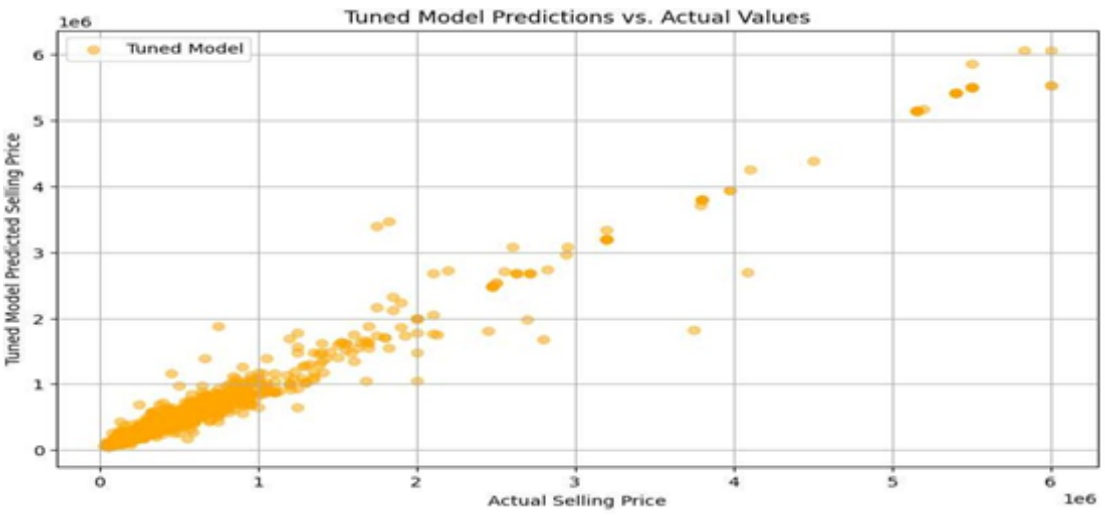
improvement in R-squared suggest that the tuning process succeeded in refining the model's predictive capabilities. While there is a slight increase in MSE, the overall benefits in accuracy and goodness of fit justify the hyperparameter tuning effort.

When considering computational cost as well, Hyperparameter tuning proves to be useful.
memory and time profiling for hyperparameter tuned model
Peak Memory: 494.09 MiB (Increment: 0.00 MiB)
Processing Time: 47.4 ms ± 4.19 ms per loop

The computational cost of the hyperparameter-tuned model remained low, with almost the same amount of memory usage but lower processing time.
Therefore, the hyperparameter-tuned Random Forest model can be considered the best choice, as it creates a balance between accuracy and computational efficiency. The refined model showcases improved predictive capabilities, making it a valuable asset for predicting vehicle selling prices.

Metrics/ Resource utilization	Random Forest	Hyperparameter tuning
Mean Absolute Error (MAE)	75127.64270478013	71639.18430903145
Mean Squared Error (MSE)	23472129077.747635	24550121004.2247
R-squared (R^2)	0.9647619457564942	0.9662111286555359
Memory Usage	485.51 MiB	494.09 MiB
Time Utilisation	88.6 ms ± 32.9 ms per loop	47.4 ms ± 4.19 ms per loop

Data visualisation for Hyperparameter tuning also looks similar to random forest due to minute changes in the errors in predictions.



7. Conclusion

In this analysis of predicting vehicle selling prices based on various features, we experimented on baseline models to advanced techniques, exploring feature engineering and hyperparameter tuning.

Baseline Models: For simpler models, K-Nearest Neighbors (KNN) and the linear regression were used as baseline models. KNN proved to be the more promising baseline model after a thorough study, exhibiting better results in terms of error metrics and R-squared.

Advanced Models: Random Forest and Gradient Boosting were chosen for their ability to handle non-linear relationships. Both models outperformed the baseline models, according to the study, with Random Forest showing somewhat better accuracy and efficiency.

Feature engineering: Two methods, automatic and manual, were considered. At first there we considered features to improve the Random Forest model, However both manual and automatic feature engineering demonstrated minimal improvements in predictive accuracy. It appeared that the original attributes were optimal to identify underlying patterns in the data.

Hyperparameter tuning: To optimise model performance, hyperparameter tuning was performed on the Random Forest model. The adjusted model showed improved accuracy and goodness of fit, as evidenced by a significant decrease in Mean Absolute Error (MAE) and an increase in R-squared.

The comparison between the baseline, tuned, and feature-engineered models revealed that the hyperparameter-tuned Random Forest model can be considered the best model for this problem. Despite a marginal increase in Mean Squared Error (MSE), the tuned model showed a better balance between accuracy and computational efficiency.

8. References

- Vehicle dataset. (2023, January 14). Kaggle. https://www.kaggle.com/datasets/nehalbirla/vehicle-dataset-from-cardekho?select=Car+details+v3.csv

- Joshi, P. (2022, July 26). A Hands-On Guide to Automated Feature Engineering using Feature Tools in Python. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2018/08/guide-automated-feature-engineering-featuretools-python/

- Brownlee, J. (2020, September 18). Hyperparameter optimization with random search and grid search. MachineLearningMastery.com. https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/