**Cell 1: Importing Required Libraries**

* **Purpose: Importing the necessary libraries for data analysis and visualization.**
* **Description: This code cell imports the Style class from the colorama library. The Style class allows for the customization of text styles in the console output, such as brightness, colors, and background. The imported Style class is used later in the code.**

**Cell 2: Calculating and Printing Statistics about Car Prices**

* **Purpose: Analyzing car prices and printing statistics.**
* **Description: This code cell calculates and prints various statistics related to car prices. It defines a list price\_thresholds that contains different price thresholds. Then, in a loop, it iterates over each threshold and counts the number of cars in the DataFrame df1 that have a price higher than the threshold. The count is stored in the variable count\_above\_threshold and printed using print(). After the loop, mean, median, minimum, and maximum prices are calculated from the 'price' column of df1 and printed using formatted strings and the print() function.**

**Cell 3: Displaying DataFrame Head**

* **Purpose: Displaying the head of the DataFrame.**
* **Description: This code cell displays the first few rows of the DataFrame df1 using the head() method. The head of the DataFrame shows a preview of the data, including columns such as 'name', 'seller', 'offerType', 'price', 'abtest', 'vehicleType', 'yearOfRegistration', 'gearbox', 'powerPS', 'model', 'kilometer', 'monthOfRegistration', 'fuelType', 'brand', 'notRepairedDamage', 'dateCreated', and 'postalCode'.**

**Cell 4: Calculating Car Age**

* **Purpose: Calculating the age of cars.**
* **Description: This code cell calculates the age of cars by subtracting the 'yearOfRegistration' column in df1 from the variable current\_year, which is set to 2023. The calculated age is assigned to a new column called 'age' in df1.**

**Cell 5: Checking for Missing Data**

* **Purpose: Checking the percentage of missing data in the DataFrame.**
* **Description: This code cell calculates the percentage of missing data in each column of df1 using the isna().sum() method to count the number of missing values and then dividing it by the length of df1. The result is multiplied by 100 to obtain the percentage of missing values for each column. The output shows the column names and the corresponding percentages.**

**Cell 6: Data Cleaning**

* **Purpose: Cleaning the dataset by removing rows and columns.**
* **Description: This code cell performs data cleaning operations on df1. It drops rows where the 'price' column values are not within the range of 200 to 20,000 using the between() method. Next, it drops rows where the 'powerPS' column values are either less than or equal to 0 or greater than 1000. Finally, it drops unnecessary columns such as 'postalCode', 'dateCreated', 'name', 'monthOfRegistration', 'yearOfRegistration', 'seller', and 'offerType' using the drop() method with axis=1 and inplace=True.**

**Cell 7: Filtering Fuel Types**

* **Purpose: Filtering out rows with a specific fuel type.**
* **Description: This code cell filters out rows in df1 where the 'fuelType' column values are equal to 'Other'. The filtered rows are removed from the DataFrame.**

# Import the necessary libraries

from sklearn.preprocessing import LabelEncoder, StandardScaler

from sklearn.tree import DecisionTreeRegressor

from sklearn.model\_selection import GridSearchCV, train\_test\_split, KFold

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

# Initialize the LabelEncoder

le = LabelEncoder()

# Transform categorical features using LabelEncoder

for categorical\_feature in categorical\_features:

df1[categorical\_feature] = le.fit\_transform(df1[categorical\_feature])

# Display the head of the transformed DataFrame

df1.head()

# Split the data into features (x) and target variable (y)

x = df1.drop(['price'], axis=1)

y = df1['price']

# Initialize the StandardScaler

scaler = StandardScaler()

# Standardize the numerical features using StandardScaler

numerical\_features = ['kilometer', 'powerPS', 'age']

x[numerical\_features] = scaler.fit\_transform(x[numerical\_features])

# Display the head of the standardized DataFrame

x.head()

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=15)

# Create a base Decision Tree Regressor

base\_model = DecisionTreeRegressor()

# Fit the base model on the training data

base\_model.fit(X\_train, y\_train)

# Predict the target variable for the test data

y\_pred = base\_model.predict(X\_test)

# Calculate the R² score

r2\_value = r2\_score(y\_test, y\_pred)

print(f'R² Score: {r2\_value:.4f}')

# Perform hyperparameter tuning using Grid Search for Decision Tree Regressor

param\_grid = {

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4]

}

# Create a GridSearchCV object

grid\_search = GridSearchCV(estimator=base\_model, param\_grid=param\_grid, scoring='neg\_mean\_squared\_error', cv=5)

# Fit the GridSearchCV object on the training data

grid\_search.fit(X\_train, y\_train)

# Get the best hyperparameters

best\_params = grid\_search.best\_params\_

# Train the model with the best hyperparameters

best\_model = grid\_search.best\_estimator\_

best\_model.fit(X\_train, y\_train)

# Predict the target variable for the test data using the best model

y\_pred = best\_model.predict(X\_test)

# Calculate evaluation metrics (MAE, MSE)

mae = mean\_absolute\_error(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

# Perform cross-validation to assess model stability

kf = KFold(n\_splits=5, shuffle=True, random\_state=42)

# Print the evaluation metrics

print(f"Best Hyperparameters: {best\_params}")

print(f"Mean Absolute Error: {mae:.2f}")

print(f"Mean Squared Error: {mse:.2f}")

# Calculate the R² score for the best model

r2\_value = r2\_score(y\_test, y\_pred)

print(f'R² Score: {r2\_value:.4f}')

# Calculate the residuals (differences between actual and predicted values)

residuals = y\_test - y\_pred

# Create a DataFrame for visualization

df\_res = pd.DataFrame({'Actual': y\_test, 'Predicted': y\_pred, 'Residuals': residuals})

# Create a heatmap for residuals

sns.heatmap(df\_res.corr(), annot=True, cmap='coolwarm', fmt=".2f")

# Add labels and title to the heatmap

plt.title('Correlation Matrix Heatmap for Regression Residuals')

plt.show()

This code snippet performs the following steps:

Import the necessary libraries.

Initialize the LabelEncoder and transform categorical features in the DataFrame.

Split the data into features (x) and target variable (y).

Initialize the StandardScaler and standardize the numerical features in the DataFrame.

Split the data into training and testing sets.

Create a base Decision Tree Regressor and fit it on the training data.

Predict the target variable for the test data and calculate the R² score.

Perform hyperparameter tuning using Grid Search for the Decision Tree Regressor.

Get the best hyperparameters and train the model with the best hyperparameters.

Predict the target variable for the test data using the best model.

Calculate evaluation metrics (MAE, MSE) and perform cross-validation.

Print the evaluation metrics and the R² score for the best model.

Calculate theresiduals (differences between actual and predicted values) and create a DataFrame for visualization.

Create a heatmap to visualize the correlation between the actual, predicted, and residual values.

Display the heatmap.  
To create a Random Forest Regressor model, you can use the following code:

python

Copy

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import r2\_score, mean\_absolute\_error, mean\_squared\_error

# Create a Random Forest Regressor model

rf\_model = RandomForestRegressor(n\_estimators=100, random\_state=42)

# Train the model

rf\_model.fit(X\_train, y\_train)

# Predict on the test set

y\_pred = rf\_model.predict(X\_test)

# Evaluate the model

r2 = r2\_score(y\_test, y\_pred)

MAEValue = mean\_absolute\_error(y\_test, y\_pred)

MSEValue = mean\_squared\_error(y\_test, y\_pred)

print('Mean Absolute Error Value is:', MAEValue)

print('Mean Squared Error Value is:', MSEValue)

print(f'R² Score: {r2:.4f}')

To tune the hyperparameters of the Random Forest Regressor model using RandomizedSearchCV, you can use the following code:

python

Copy

from sklearn.model\_selection import RandomizedSearchCV

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import r2\_score, mean\_absolute\_error, mean\_squared\_error

# Create a Random Forest Regressor model

rf\_model = RandomForestRegressor(random\_state=42)

# Define the hyperparameter distributions

param\_distributions = {

'n\_estimators': [50, 100, 150],

'max\_depth': [None, 10, 20],

'min\_samples\_split': [2, 5, 10]

}

# Create the RandomizedSearchCV object

random\_search = RandomizedSearchCV(

estimator=rf\_model,

param\_distributions=param\_distributions,

n\_iter=100,

scoring='r2',

cv=5

)

# Fit the RandomizedSearchCV object to the training data

random\_search.fit(X\_train, y\_train)

# Get the best hyperparameters

best\_params = random\_search.best\_params\_

# Train the model with the best hyperparameters

best\_rf\_model = RandomForestRegressor(random\_state=42, \*\*best\_params)

best\_rf\_model.fit(X\_train, y\_train)

# Predict on the test set

y\_pred\_tuned = best\_rf\_model.predict(X\_test)

# Evaluate the tuned model

r2\_tuned = r2\_score(y\_test, y\_pred\_tuned)

MAEValue\_tuned = mean\_absolute\_error(y\_test, y\_pred\_tuned)

MSEValue\_tuned = mean\_squared\_error(y\_test, y\_pred\_tuned)

# Print the results

print('Best Hyperparameters:', best\_params)

print('Tuned R² Score:', r2\_tuned)

print('Tuned Mean Absolute Error:', MAEValue\_tuned)

print('Tuned Mean Squared Error:', MSEValue\_tuned)

To create an Extra Trees Regressor model, you can use the following code:

python

Copy

from sklearn.ensemble import ExtraTreesRegressor

from sklearn.metrics import r2\_score, mean\_absolute\_error, mean\_squared\_error

# Create an Extra Trees Regressor model

et\_model = ExtraTreesRegressor(n\_estimators=100, random\_state=42)

# Train the model

et\_model.fit(X\_train, y\_train)

# Predict on the test set

y\_pred\_et = et\_model.predict(X\_test)

# Evaluate the model

r2\_et = r2\_score(y\_test, y\_pred\_et)

MAE\_et = mean\_absolute\_error(y\_test, y\_pred\_et)

MSE\_et = mean\_squared\_error(y\_test, y\_pred\_et)

# Print the evaluation metrics

print('Extra Trees R² Score:', r2\_et)

print('Extra Trees Mean Absolute Error:', MAE\_et)

print('Extra Trees Mean Squared Error:', MSE\_et)

To tune the hyperparameters of the Extra Trees Regressor model using GridSearchCV, you can use the following code:

python

Copy

from sklearn.model\_selection import GridSearchCV

from sklearn.ensemble import ExtraTreesRegressor

from sklearn.metrics import r2\_score, mean\_absolute\_error, mean\_squared\_error

# Define the parameter grid for tuning

param\_grid\_et = {

'n\_estimators': [50, 100, 150],

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4]

}

# Create an Extra Trees Regressor model

et\_model = ExtraTreesRegressor(random\_state=42)

# Instantiate GridSearchCV

grid\_search\_et = GridSearchCV(et\_model, param\_grid\_et, cv=5, scoring='r2', n\_jobs=-1)

# Fit the grid search to the data

grid\_search\_et.fit(X\_train, y\_train)

# Get the best hyperparameters

best\_params\_et = grid\_search\_et.best\_params\_

# Train the model with the best hyperparameters

best\_et\_model = ExtraTreesRegressor(\*\*best\_params\_et, random\_state=42)

best\_etIn the above code, `X\_train` and `y\_train` represent the training features and target values, respectively. `X\_test` is the test feature data.

 model selection and hyperparameter tuning using scikit-learn's Pipeline, GridSearchCV, and cross\_val\_score functions. Let's break down the code and write documentation for it:

python

Copy

from sklearn.pipeline import Pipeline

from sklearn.model\_selection import GridSearchCV, cross\_val\_score

from sklearn.neighbors import KNeighborsRegressor

from xgboost import XGBRegressor

from catboost import CatBoostRegressor

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import make\_scorer

The code imports necessary modules and classes from scikit-learn and other libraries.

python

Copy

# Define models

models = {

'KNN': KNeighborsRegressor(),

'XGBoost': XGBRegressor(),

'CatBoost': CatBoostRegressor(silent=True),

}

In this section, the code defines three regression models: K-Nearest Neighbors (KNN), XGBoost, and CatBoost. The models are initialized without any hyperparameters.

python

Copy

# Create pipelines

pipelines = {}

for name, model in models.items():

pipelines[name] = Pipeline([

('scaler', StandardScaler()), # Include standard scaler in the pipeline

('model', model),

])

Here, the code creates pipelines for each model. Each pipeline consists of two steps: scaling the features using StandardScaler and applying the corresponding model. The pipelines are stored in a dictionary called pipelines, with the model names as keys.

python

Copy

# Define hyperparameter grids

param\_grids = {

'KNN': {'model\_\_n\_neighbors': [3, 5, 7, 9]},

'XGBoost': {'model\_\_n\_estimators': [50, 100, 200], 'model\_\_learning\_rate': [0.01, 0.1, 0.2]},

'CatBoost': {'model\_\_iterations': [50, 100, 200], 'model\_\_learning\_rate': [0.01, 0.1, 0.2]},

}

In this section, the code defines hyperparameter grids for each model. Each grid contains the hyperparameters to be tuned along with their corresponding values. The hyperparameters are specified using the pipeline syntax, where the double underscore (\_\_) separates the step name (model) from the hyperparameter name.

python

Copy

# Define scoring metrics for GridSearchCV and cross-validation

scoring\_metric\_mse = make\_scorer(mean\_squared\_error, greater\_is\_better=False)

scoring\_metric\_r2 = make\_scorer(r2\_score)

Here, the code defines two scoring metrics: mean squared error (MSE) and R2 score. The make\_scorer function is used to create scorer objects from the corresponding scikit-learn metrics.

python

Copy

# Hyperparameter tuning using GridSearchCV

results = {}

for name, pipeline in pipelines.items():

param\_grid = param\_grids[name]

# GridSearchCV for Mean Squared Error

grid\_search\_mse = GridSearchCV(pipeline, param\_grid, cv=5, scoring=scoring\_metric\_mse, n\_jobs=-1)

grid\_search\_mse.fit(X\_train, y\_train)

# GridSearchCV for R2 score

grid\_search\_r2 = GridSearchCV(pipeline, param\_grid, cv=5, scoring=scoring\_metric\_r2, n\_jobs=-1)

grid\_search\_r2.fit(X\_train, y\_train)

results[name] = {

'Best Hyperparameters': grid\_search\_mse.best\_params\_,

'Best Mean Squared Error': -grid\_search\_mse.best\_score\_,

'Best R2 Score': grid\_search\_r2.best\_score\_,

}

This section performs hyperparameter tuning using GridSearchCV for each model. For each model, a grid search is conducted with cross-validation (cv=5) to find the best hyperparameters that minimize the mean squared error (MSE) and maximize the R2 score. The results are stored in the results dictionary, which contains the best hyperparameters, best MSE, and best R2 score for each model.

python

Copy

# Display results

for name, result in results.items():

print(f"\nModel: {name}")

print(f"Best Hyperparameters: {result['Best Hyperparameters']}")

print(f"Best Mean Squared Error: {result['Best Mean Squared Error']}")

print(f"Best R2 Score: {result['Best R2 Score']}")

print("---------------------------")

In this section, the code displays the results of hyperparameter tuning for each model. It prints the model name, best hyperparameters, best MSE, and best R2 score.

python

Copy

# Evaluate accuracy for each model using cross-validation

for name, pipeline in pipelines.items():

# Cross-validation for Mean Squared Error

scores\_m```python

# Evaluate accuracy for each model using cross-validation

for name, pipeline in pipelines.items():

# Cross-validation for Mean Squared Error

scores\_mse = cross\_val\_score(pipeline, X\_train, y\_train, cv=5, scoring='neg\_mean\_squared\_error')

mean\_mse = -scores\_mse.mean()

# Cross-validation for R2 score

scores\_r2 = cross\_val\_score(pipeline, X\_train, y\_train, cv=5, scoring='r2')

mean\_r2 = scores\_r2.mean()

print(f"\nModel: {name}")

print(f"Mean Squared Error (Cross-Validation): {mean\_mse}")

print(f"R2 Score (Cross-Validation): {mean\_r2}")

print("---------------------------")

In this section, the code evaluates the performance of each model using cross-validation. For each model, it calculates the mean squared error (MSE) and R2 score using the cross\_val\_score function with 5-fold cross-validation. The negative MSE scores are converted to positive values for easier interpretation. The mean MSE and mean R2 score are then printed for each model.

This code snippet provides a complete example of model selection and hyperparameter tuning using scikit-learn's Pipeline, GridSearchCV, and cross\_val\_score functions. It demonstrates how to create pipelines, define hyperparameter grids, perform grid search with cross-validation, and evaluate model performance. The results are displayed for each model, including the best hyperparameters, best MSE, and best R2 score.  
The code you provided is using the PyCaret library for regression analysis. Here's a breakdown of what the code does:

1. exp1 = setup(df1, target='price', session\_id=123): This line sets up the PyCaret experiment by specifying the dataset df1 and the target variable price to be predicted. The session\_id parameter is set to 123 for reproducibility.
2. best\_model = compare\_models(): This line compares and evaluates multiple regression models available in PyCaret and selects the best performing model based on a default evaluation metric (usually R-squared). The best model is stored in the best\_model variable.
3. final\_model = create\_model(best\_model): This line creates the final model using the best performing algorithm identified in the previous step. The model is stored in the final\_model variable.
4. evaluate\_model(final\_model): This line evaluates the performance of the final model using various regression evaluation metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), R-squared (R2), Root Mean Squared Logarithmic Error (RMSLE), Mean Absolute Percentage Error (MAPE), and Time Taken for training (TT).

The output you provided includes information about the setup, data shapes, feature types, and evaluation metrics for each model compared. It also shows the evaluation metrics for each fold in the cross-validation process.