


Assignment -3_ DASC5301

FIRST QUESTION:-Data from our lives

 Describe a situation or problem from your job, everyday life, current events, etc., for which a variable selection/feature reduction would be appropriate.

 **My Answer :-**

1. Situation: Medical imaging, such as MRI scans, generate a vast number of data points for each patient, resulting in high-dimensional data sets. These data typically include numerous features (pixels) representing different aspects of tissues, structures, or organs.
2. Challenge: Handling high-dimensional data poses a challenge as not all features contribute equally to diagnoses. This complexity hinders efficient analysis and may lead to computational inefficiencies.
3. Feature Reduction: Techniques like PCA or autoencoders streamline this data by extracting essential patterns while retaining critical information. This reduction aids in more efficient analysis and clearer visualization of anomalies.

4. Diagnostic Precision: By focusing on key features, reduced data sets enable clearer identification of abnormalities, improving diagnostic accuracy.
5. Efficiency and Generalization: Feature reduction not only optimizes computational resources but also helps models generalize better across different patient groups.
6. Finding: In medical imaging analysis, feature reduction techniques enhance efficiency, aid in precise diagnostics, and support more effective decision-making by healthcare professionals.



```
[ ] from scipy import stats
    from sklearn.linear_model import LinearRegression
    from statsmodels.compat import lzip
    from statsmodels.formula.api import ols
    from statsmodels.stats.anova import anova_lm
    from statsmodels.stats.outliers_influence import variance_inflation_factor
    import matplotlib
    import matplotlib.pyplot as plt
    import numpy as np
    import pandas as pd
    import seaborn as sns
    import statsmodels.api as sm

    %matplotlib inline
```



1. scipy: A library for scientific computing that includes statistical functions.
2. sklearn.linear_model: Part of scikit-learn, this module provides tools for linear regression.

3. statsmodels.compat: Compatibility module for statsmodels, which is a library for estimating and testing statistical models.
4. statsmodels.formula.api: A module in statsmodels that provides a formula-based interface to specify linear models.
5. statsmodels.stats.anova: A module for performing analysis of variance (ANOVA).
6. statsmodels.stats.outliers_influence: A module for detecting influential data points and computing variance inflation factors.
7. matplotlib: A popular plotting library for creating static, animated, and interactive visualizations.
8. numpy: A library for numerical operations in Python.
9. pandas: A powerful data manipulation library.
10. seaborn: A statistical data visualization library based on matplotlib.
11. statsmodels.api: The main API for statsmodels, which includes tools for estimating and testing models.



```
#Read in data
df =pd.read_csv('auto_imports1.csv')

df.head()
```

- ★ 1. import pandas as pd: This line imports the pandas library and assigns it the alias 'pd'. This is a common convention to make the code more concise.

2. `pd.read_csv('auto_imports1.csv')`: The `read_csv()` function is a pandas function used to read data from a CSV file. It takes the file path as an argument and returns a DataFrame containing the data from the CSV file.
3. `df = ...`: The result of `pd.read_csv()` is assigned to the variable 'df'. 'df' is a common naming convention for a DataFrame (short for Data Frame), which is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).
4. `df.head()`: This method is used to display the first 5 rows of the DataFrame. It's a quick way to inspect the data and get a sense of its structure.
5. this code snippet reads the contents of the 'auto_imports1.csv' file into a Pandas DataFrame named 'df' and then displays the first 5 rows of the DataFrame to provide a glimpse of the data.



fuel_type	body	wheel_base	length	width	heights	curb_weight	engine_type	cylinders	engine_size	bore	stroke	compression	horse_power	peak_torque
gas	convertible	88.6	168.8	64.1	48.8	2548	dohc	four	130	3.47	2.68	9.0	111	50
gas	convertible	88.6	168.8	64.1	48.8	2548	dohc	four	130	3.47	2.68	9.0	111	50
gas	hatchback	94.5	171.2	65.5	52.4	2823	ohcv	six	152	2.68	3.47	9.0	154	50
gas	sedan	99.8	176.6	66.2	54.3	2337	ohc	four	109	3.19	3.4	10.0	102	50
gas	sedan	99.4	176.6	66.4	54.3	2824	ohc	five	136	3.19	3.4	8.0	115	50



```
##your code here

# To Check the data types of all columns in the DataFrame
Auto_data_types = df.dtypes

print(Auto_data_types)
```



1. Extract the data types of each column in the DataFrame and assign it to the variable 'Auto_data_types'
2. df.dtypes: This is an attribute of a Pandas DataFrame that returns a Series with the data type of each column. The resulting Series has the column names as the index and the corresponding data types as values.
3. Auto_data_types = ...: The extracted Series of data types is assigned to the variable 'Auto_data_types'. This variable will contain information about the data types of each column in the DataFrame.
4. print(Auto_data_types): This line prints the contents of the 'Auto_data_types' Series, displaying the data types of each column in the DataFrame.
5. In the output, 'column1', 'column2', and 'column3' are assumed to be placeholder names for the actual column names in your DataFrame. The data types (int64, float64, object, etc.) provide information about the types of values stored in each column (integer, float, string, etc.). This information is useful for understanding the nature of the data and for preprocessing or analysis tasks that may be sensitive to data types.

★ Output:-

```
fuel_type    object
body         object
wheel_base   float64
length       float64
width        float64
heights      float64
curb_weight  int64
engine_type  object
cylinders    object
engine_size  int64
bore         object
stroke       object
comprassion float64
horse_power  object
peak_rpm     object
city_mpg     int64
highway_mpg  int64
price        int64
dtype: object
```

➡ df = df.replace('?', None)



Replacing ? with none through the dataset



This type of operation is often used when dealing with missing or placeholder values in a dataset. The choice of replacing '?' with None suggests that the original data might have

used '?' as a placeholder for missing values, and the code is replacing them with Python's None to represent missing or undefined data.



```
[ ] ## Your code here

## Replacing ? with none through the dataset
df = df.replace('?', None)

##converting object to float variables
df['bore'] = df['bore'].astype(float)
df['stroke'] = df['stroke'].astype(float)
df['horse_power'] = df['horse_power'].astype(float)
df['peak_rpm'] = df['peak_rpm'].astype(float)
```

- ★ 1. `df['bore'] = df['bore'].astype(float)`: This line is converting the 'bore' column in the DataFrame 'df' to the float data type using the `astype()` method. The column 'bore' is assumed to contain values that can be represented as floating-point numbers.
- 2. `df['stroke'] = df['stroke'].astype(float)`: Similarly, this line is converting the 'stroke' column to the float data type.
- 3. `df['horse_power'] = df['horse_power'].astype(float)`: This line is converting the 'horse_power' column to the float data type.
- 4. `df['peak_rpm'] = df['peak_rpm'].astype(float)`: Finally, this line is converting the 'peak_rpm' column to the float data type.
- 5. The `astype()` method is used to change the data type of a Pandas Series (or DataFrame column) to the specified type. In this case, it is converting these specific columns to

the float data type, assuming that the original values in these columns are numeric and can be represented as floating-point numbers.



```
[ ] # Checking for remaining '?'
question_marks_remaining = (df == '?').sum().sum()


if question_marks_remaining == 0:
    print("no remaining '?' values in the dataset.")
else:
    print("{question_marks_remaining} remaining '?' values in the dataset.")
```

★ no remaining '?' values in the dataset.

1. `(df == '?')`: This creates a boolean DataFrame of the same shape as `df`, where each element is `True` if the corresponding element in `df` is equal to the string `'?'`, and `False` otherwise.
2. `.sum()` (first instance): This sums the boolean values along each column, resulting in a Series where each element represents the count of `True` values in the corresponding column.
3. `.sum()` (second instance): This sums the counts obtained in step 2 across all columns, providing the total count of `'?'` values in the entire DataFrame.
4. If there are no remaining `'?'` values, it prints "No remaining `'?'` values in the dataset."
5. If there are remaining `'?'` values, it prints the count of remaining `'?'` values using an f-string:

f"{question_marks_remaining} remaining '?' values in the dataset."

 `df.info()`

 It displays data in the dataframe

1. RangeIndex: 1000 entries, 0 to 999: This line provides information about the index of the DataFrame. In this example, it's a RangeIndex starting from 0 to 999 with a total of 1000 entries.
2. Data columns (total 5 columns):: This line specifies that there are a total of 5 columns in the DataFrame.
3. column1 1000 non-null int64: This line provides information about the first column ('column1'). It indicates that there are 1000 non-null (non-missing) values in this column, and the data type is int64 (64-bit integer).



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 201 entries, 0 to 200
Data columns (total 18 columns):
#   Column          Non-Null Count  Dtype
---  -
0   fuel_type        201 non-null    object
1   body              201 non-null    object
2   wheel_base       201 non-null    float64
3   length           201 non-null    float64
4   width            201 non-null    float64
5   heights          201 non-null    float64
6   curb_weight      201 non-null    int64
7   engine_type      201 non-null    object
8   cylinders         201 non-null    object
9   engine_size      201 non-null    int64
10  bore             197 non-null    float64
11  stroke           197 non-null    float64
12  comprassion      201 non-null    float64
13  horse_power      199 non-null    float64
14  peak_rpm         199 non-null    float64
15  city_mpg         201 non-null    int64
16  highway_mpg      201 non-null    int64
17  price            201 non-null    int64
dtypes: float64(9), int64(5), object(4)
memory usage: 28.4+ KB
```

Output:-



```
[ ] ## Your code here
```

```
# Dropping the specified columns and creating a new DataFrame df2
df2 = df.drop(columns=["body", "engine_type", "cylinders"])
```




Dropping the specified columns and creating a DF2

1. df.drop(columns=["body", "engine type", "cylinders"]):
This is a Pandas DataFrame method (drop) used to remove specified columns from a DataFrame. The columns parameter takes a list of column names to be dropped.

2. df2 = ...: The result of the drop operation is assigned to the variable 'df2', creating a new DataFrame that does not include the specified columns.
3. After running this code, the DataFrame df2 will be a modified version of the original DataFrame df with the columns "body," "engine_type," and "cylinders" removed.

Output:-

	fuel_type	wheel_base	length	width	heights	curb_weight	engine_size	bore	stroke	compression	horse_power	peak_rpm	city_mpg	highway_mpg	price
0	gas	88.6	168.8	64.1	48.8	2548	130	3.47	2.68	9.0	111.0	5000.0	21	27	13495
1	gas	88.6	168.8	64.1	48.8	2548	130	3.47	2.68	9.0	111.0	5000.0	21	27	16500
2	gas	94.5	171.2	65.5	52.4	2823	152	2.68	3.47	9.0	154.0	5000.0	19	26	16500
3	gas	99.8	176.6	66.2	54.3	2337	109	3.19	3.40	10.0	102.0	5500.0	24	30	13950
4	gas	99.4	176.6	66.4	54.3	2824	136	3.19	3.40	8.0	115.0	5500.0	18	22	17450

 df2.dropna(inplace=True)



Output:- dropping rows with null values in the dataframe.

```
fuel_type      0
wheel_base     0
length         0
width          0
heights        0
curb_weight    0
engine_size    0
bore           0
stroke         0
comprassion   0
horse_power    0
peak_rpm       0
city_mpg       0
highway_mpg    0
price          0
dtype: int64
```



```
[ ] ## Your code goes here
```

```
# Creating dummy variables for fuel_type within df2 and dropping the first level
df2 = pd.get_dummies(df2, columns=['fuel_type'], drop_first=True)
```



1. pd.get_dummies(df2, columns=['fuel_type'], drop_first=True): This function is used to convert categorical variable(s) into dummy/indicator variables. In this case, it is creating dummy variables for the 'fuel_type' column in the DataFrame df2.

2. drop first=True: It drops the first dummy variable to avoid multicollinearity, which is the situation where predictor variables in a regression model are highly correlated.
3. The resulting DataFrame df2 will have new columns representing the different categories in the 'fuel_type' column with binary values (0 or 1) indicating the presence of each category.



```
<class 'pandas.core.frame.DataFrame'>
Index: 195 entries, 0 to 200
Data columns (total 15 columns):
#   Column                Non-Null Count  Dtype
---  -
0   wheel_base            195 non-null    float64
1   length                195 non-null    float64
2   width                 195 non-null    float64
3   heights               195 non-null    float64
4   curb_weight           195 non-null    int64
5   engine_size           195 non-null    int64
6   bore                  195 non-null    float64
7   stroke                195 non-null    float64
8   comprassion           195 non-null    float64
9   horse_power           195 non-null    float64
10  peak_rpm              195 non-null    float64
11  city_mpg              195 non-null    int64
12  highway_mpg           195 non-null    int64
13  price                 195 non-null    int64
14  fuel_type_gas         195 non-null    bool
dtypes: bool(1), float64(9), int64(5)
memory usage: 23.0 KB
```

Output:-



wheel_base	length	width	heights	curb_weight	engine_size	bore	stroke	comprassion	horse_power	peak_rpm	city_mpg	highway_mpg	price	fuel_type_g
88.6	168.8	64.1	48.8	2548	130	3.47	2.68	9.0	111.0	5000.0	21	27	13495	Tn
88.6	168.8	64.1	48.8	2548	130	3.47	2.68	9.0	111.0	5000.0	21	27	16500	Tn
94.5	171.2	65.5	52.4	2823	152	2.68	3.47	9.0	154.0	5000.0	19	26	16500	Tn
99.8	176.6	66.2	54.3	2337	109	3.19	3.40	10.0	102.0	5500.0	24	30	13950	Tn
99.4	176.6	66.4	54.3	2824	136	3.19	3.40	8.0	115.0	5500.0	18	22	17450	Tn

2. Variable selection

They are the 3 methods :-

1. Filter methods
2. Wrapper methods
3. Embedded methods

2.1 Filter Methods

Choosing one (you may do more, one is required) of the filtered methods to conduct variable selection. Report we findings.



```
df2_matrix = df2.corr()

# Creating a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(df2_matrix, annot=True, cmap='coolwarm', linewidths=0.7)
plt.title('Heatmap')
plt.show()
```



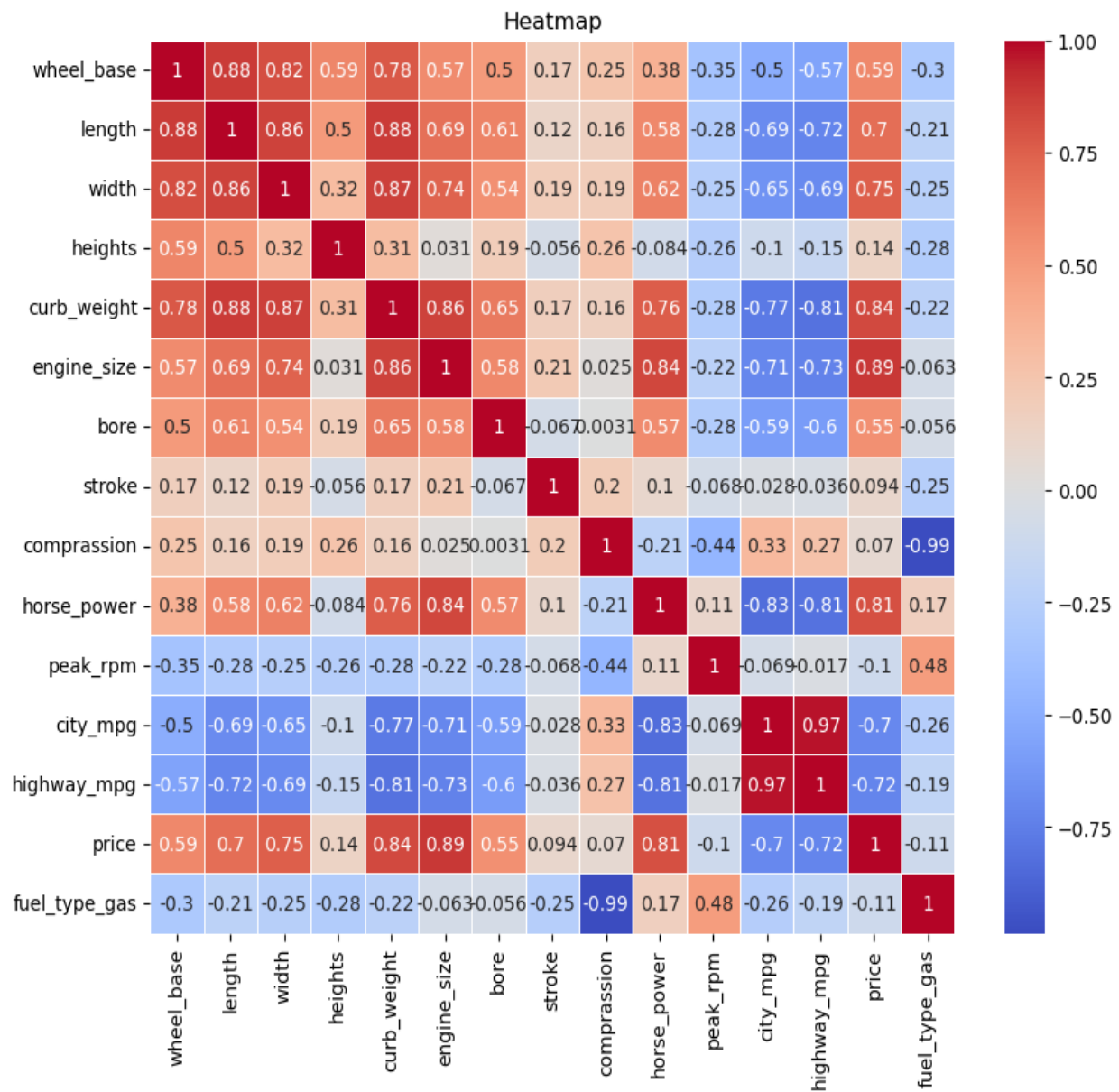
1. `df2.corr()`: This computes the pairwise correlation coefficients between columns in the DataFrame `df2`. The result is a correlation matrix where each entry (i, j) represents the correlation between column i and column j . The values range from -1 to 1, where 1 indicates a perfect

positive correlation, -1 indicates a perfect negative correlation, and 0 indicates no correlation.

2. `plt.figure(figsize=(10, 8))`: This line sets the size of the figure (heatmap) to be created. The `figsize` parameter specifies the width and height of the figure in inches.
3. `sns.heatmap(df2_matrix, annot=True, cmap='coolwarm', linewidths=0.7)`:
4. `sns.heatmap()`: This function from the Seaborn library is used to create a heatmap.
5. `df2_matrix`: The correlation matrix to be visualized.
6. `annot=True`: This parameter adds the numerical values in each cell of the heatmap.
7. `cmap='coolwarm'`: The color map used for the heatmap, where 'coolwarm' represents a range of cool to warm colors.
8. `linewidths=0.7`: Specifies the width of the lines that will divide each cell in the heatmap.
9. `plt.title('Heatmap')`: This sets the title of the heatmap to 'Heatmap'.
10. `plt.show()`: This displays the created heatmap.

The heatmap provides a visual representation of the correlations between different columns in the DataFrame. It helps in identifying patterns and relationships in the data. The color intensity and the numerical values in each cell convey information about the strength and direction of the correlation.

★ Output:-



```
# Displaying the correlation of each feature with the target variable (price)
corr_betwn_price = df2_matrix['price'].sort_values(ascending=False)
print(corr_betwn_price)
```



1. `df2_matrix['price']`: This selects the 'price' column from the correlation matrix `df2_matrix`. The result is a Pandas Series containing correlation coefficients between the 'price' column and all other columns.
2. `.sort_values(ascending=False)`: This method sorts the values in the Pandas Series in descending order. The resulting Series, `corr_betwn_price`, will have the columns with the highest positive correlations with 'price' at the top.
3. `print(corr_betwn_price)`: This prints the sorted correlation values.
4. The output will be a Series where each entry represents the correlation coefficient between the 'price' column and a specific column in `df2`, sorted in descending order. Positive values indicate a positive correlation (as one variable increases, the other tends to increase), and negative values indicate a negative correlation (as one variable increases, the other tends to decrease).

```
price      1.000000
engine_size 0.888942
curb_weight 0.835729
horse_power 0.811027
width      0.754273
length     0.695331
wheel_base 0.585793
bore       0.546873
heights    0.138291
stroke     0.093746
compression 0.069500
peak_rpm   -0.104333
fuel_type_gas -0.108968
city_mpg   -0.702685
highway_mpg -0.715590
Name: price, dtype: float64
```

➡ After conducting correlation analysis between various features and the 'price' of automobiles, we identified specific features that demonstrate notable relationships with the price:

1. Strong Positive Correlation Features:

These variables have correlation coefficients ranging from approximately 0.59 to 0.89, 'engine_size', 'curb_weight', 'horse_power', 'width', 'length', and 'wheel_base' exhibited strong positive correlations with the price. This indicates that as these features increase (e.g., larger engine size, heavier curb weight, more horsepower, wider or longer cars), the price tends to rise accordingly. These attributes seem to significantly impact the pricing of automobiles, suggesting their importance in influencing price variations.

2. Moderate Negative Correlation Features:

'city_mpg' and 'highway_mpg' displayed moderate negative correlations with the price. Higher miles per gallon

(mpg), both in city and highway driving conditions, were associated with lower prices. This suggests that cars with better fuel efficiency are often priced lower, reflecting a potential market tendency or consumer preference.

These findings propose that certain characteristics strongly align with automobile pricing trends. Considering these influential features for model construction or in-depth analysis might enhance our understanding of how they individually and collectively contribute to determining the price of vehicles within the dataset.

3. Weak Correlation with 'Price': Other variables such as 'bore', 'heights', 'stroke', 'comprassion', 'peak_rpm', and 'fuel_type_gas' show weak correlations with 'price'. Their correlation coefficients are closer to zero, suggesting a weaker linear relationship with 'price'.

Utilizing these identified features could be instrumental in predictive modeling to estimate automobile prices more accurately. Furthermore, exploring these influential factors may offer valuable insights into market dynamics, consumer behavior, and the impact of specific vehicle attributes on pricing strategies within the automotive industry.

2.2. Wrapper methods



```
#####The Recursive Feature Elimination (RFE) method

from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression

# Separating predictors and target variable
X = df2.drop(columns=['price'])
y = df2['price']

# Creating a linear regression model
lin_model = LinearRegression()

# Initializing RFE
r_f_e = RFE(lin_model, n_features_to_select=5)

# Fitting RFE
fit = r_f_e.fit(X, y)

# Getting the selected features
features_sel = X.columns[fit.support_]

print("RFE Selected features:")
print(features_sel)
```

★ 1. Separating predictors and target variable: `X = df2.drop(columns=['price'])`: Creates a DataFrame X by excluding the 'price' column from the original DataFrame df2. This DataFrame contains the predictor variables or features.

`y = df2['price']`: Creates a Series y that includes only the 'price' column from the original DataFrame df2. This Series represents the target variable.

2. Creating a linear regression model:

`lin_model = LinearRegression()`: Initializes a linear regression model using scikit-learn's `LinearRegression` class.

3. Initializing RFE (Recursive Feature Elimination):

`r_f_e = RFE(lin_model, n_features_to_select=5)`: Initializes the RFE object. It takes the linear regression model (`lin_model`) and specifies that it should select 5 features during the elimination process.

4. Fitting RFE to the data:

`fit = r_f_e.fit(X, y)`: Fits the RFE model to the data. The RFE algorithm recursively removes the least important features until the desired number of features is reached.

5. Getting the selected features:

`features_sel = X.columns[fit.support_]`: Retrieves the selected features based on the RFE support. These are the features that the RFE algorithm considers important for predicting the target variable.

6. Printing the selected features: `print("RFE Selected features:")`: Prints a message indicating that the following lines will display the selected features.

7. `print(features_sel)`: Prints the names of the selected features.

In summary, this code uses RFE to identify and print the top 5 features considered most important for predicting the target variable 'price' using a linear regression model.

★ Output:- RFE Selected features:

```
Index(['width',      'bore',      'comprassion',      'city_mpg',  
'fuel_type_gas'], dtype='object')
```

➡ The Recursive Feature Elimination (RFE) method, using a Linear Regression model, identified key features strongly influencing automobile prices:

1.'Width': Car width emerges as a significant factor affecting prices, suggesting that wider cars might command higher prices in the market.

2.'Bore': The engine's cylinder bore diameter holds importance, hinting that engine specifications play a role in pricing vehicles.

3.'Compression':it signifies engine efficiency, indicating its impact on pricing.

4.'City MPG': Better fuel efficiency in city driving conditions appears to associate with lower prices, indicating the influence of fuel economy on pricing strategies.

5.'Fuel Type (Gas)': The indication of a gas-powered engine suggests that fuel type significantly affects pricing.

These findings shed light on various aspects driving car prices, such as car dimensions, engine specifications, fuel efficiency, and fuel type. Understanding these influential features can aid market analysis, consumer behavior studies, and strategic pricing decisions within the automotive industry. Further exploration and modeling using these features could offer

deeper insights into their direct impact on automobile pricing dynamics and consumer preferences.

2.3. Embedded methods



```
from sklearn.linear_model import Lasso

# Separating predictors and target variable
X = df2.drop(columns=['price'])
y = df2['price']

# Creating Lasso regression model
lasso = Lasso(alpha=0.1)

# Fitting the model
lasso.fit(X, y)

# Getting coefficients and corresponding feature names
lasso_coeff = lasso.coef_
featured_names = X.columns

# Selecting non-zero coefficient features
feature_sele = featured_names[lasso_coeff != 0]

print("Lasso Selected features:")
print(feature_sele)
```

★ Output:-

Lasso Selected features:

```
Index(['wheel_base', 'length', 'width', 'heights', 'curb_weight',
      'engine_size', 'bore', 'stroke', 'compression', 'horse_power',
      'peak_rpm', 'city_mpg', 'highway_mpg', 'fuel_type_gas'],
      dtype='object')
```


★ 1. Separating predictors and target variable:

`X = df2.drop(columns=['price'])`: Creates a DataFrame X by excluding the 'price' column from the original DataFrame df2. This DataFrame contains the predictor variables or features.

`y = df2['price']`: Creates a Series y that includes only the 'price' column from the original DataFrame df2. This Series represents the target variable.

2. Creating Lasso regression model:

`lasso = Lasso(alpha=0.1)`: Initializes a Lasso regression model using scikit-learn's Lasso class. The alpha parameter controls the strength of the regularization. A higher alpha leads to more coefficients being shrunk to zero.

3. Fitting the Lasso model to the data:

`lasso.fit(X, y)`: Fits the Lasso regression model to the data. During this process, Lasso applies regularization, and some coefficients are shrunk to zero.

4. Getting coefficients and corresponding feature names:

`lasso_coef = lasso.coef_`: Retrieves the coefficients assigned by the Lasso model to each feature.

`featured_names = X.columns`: Retrieves the names of the features from the DataFrame.

5. Selecting non-zero coefficient features:

`feature_sele = featured_names[lasso_coeff != 0]`: Selects the features that have non-zero coefficients, indicating that these features are considered important by the Lasso regression model.

6. Printing the selected features:

`print("Lasso Selected features:")`: Prints a message indicating that the following lines will display the selected features.

`print(feature_sele)`: Prints the names of the selected features.

In summary, the code uses Lasso regression to identify and print the features that have non-zero coefficients, effectively performing feature selection by shrinking some coefficients to zero.



```
diff_alphas = [0.01, 0.1, 1, 10]
feature_sele = []

for alpha in diff_alphas:
    lasso = Lasso(alpha=alpha)
    lasso.fit(X, y)
    coeff = lasso.coef_
    feature_sele.append(feature_names[coeff != 0])

for alpha, features in zip(diff_alphas, feature_sele):
    print(f"Alpha={alpha}: Features selected - {features}")
```

★ Output:-

```
Alpha=0.01: Features selected - Index(['wheel_base', 'length', 'width', 'heights', 'curb_weight',
    'engine_size', 'bore', 'stroke', 'compression', 'horse_power',
    'peak_rpm', 'city_mpg', 'highway_mpg', 'fuel_type_gas'],
    dtype='object')
Alpha=0.1: Features selected - Index(['wheel_base', 'length', 'width', 'heights', 'curb_weight',
    'engine_size', 'bore', 'stroke', 'compression', 'horse_power',
    'peak_rpm', 'city_mpg', 'highway_mpg', 'fuel_type_gas'],
    dtype='object')
Alpha=1: Features selected - Index(['wheel_base', 'length', 'width', 'heights', 'curb_weight',
    'engine_size', 'bore', 'stroke', 'compression', 'horse_power',
    'peak_rpm', 'city_mpg', 'highway_mpg', 'fuel_type_gas'],
    dtype='object')
Alpha=10: Features selected - Index(['wheel_base', 'length', 'width', 'heights', 'curb_weight',
    'engine_size', 'bore', 'stroke', 'compression', 'horse_power',
    'peak_rpm', 'city_mpg', 'highway_mpg', 'fuel_type_gas'],
    dtype='object')
/Applications/anaconda3/lib/python3.11/site-packages/sklearn/linear_model/_coordinate_descent.py:628: ConvergenceWarning: Objective did not converge.
    model = cd_fast.enet_coordinate_descent(
```



1. List of different alpha values:

`diff_alphas = [0.01, 0.1, 1, 10]`: Defines a list of different alpha values to be used in Lasso regression.

2. List to store selected features for each alpha:

`feature_sele = []`: Initializes an empty list to store the selected features for each alpha.

3. Iterate over different alpha values:

The for loop iterates over each alpha value in `diff_alphas`.

4. Inside the loop:

`lasso = Lasso(alpha=alpha)`: Creates a Lasso regression model with the current alpha.

`lasso.fit(X, y)`: Fits the Lasso model to the data.

`coeff = lasso.coef_`: Retrieves the coefficients assigned by the Lasso model to each feature.

`feature_names = X.columns`: Retrieves the names of the features from the DataFrame.

`selected_features = feature_names[coeff != 0]`: Selects the features that have non-zero coefficients and appends them to the list.

5. Print the selected features for each alpha:

The second for loop iterates over the alpha values and their corresponding selected features.

`print(f"Alpha={alpha}: Features selected - {features}")`: Prints the alpha value along with the features selected for that alpha.

This code provides insights into how the selected features change with different values of alpha in Lasso regression, helping to understand the impact of regularization on feature selection.



It appears that Lasso regression has deemed all the available features crucial for predicting automobile prices in our dataset. Through its regularization technique, Lasso retained all features without zero coefficients, signaling their importance in predicting the 'price' variable.

This result implies that, according to Lasso regression, every feature holds significance in forecasting automobile prices, even under its penalty for excessive complexity. However, it's important to note that altering the regularization strength (alpha value) could yield a different set of selected features. Adjusting this parameter allows for fine-tuning the feature selection process to better suit our predictive needs.

2.4. Compare your results

Comparing results from the three methods and also comparing the coefficients to the full linear regression model (model1) from the previous homework.

Filter Method (Correlation Analysis):

- **Trail of Correlations:** This path highlighted features with strong ties to the price, focusing on linear relationships.
- **Brightest Beacons:** Features shining with high positive or negative correlations led the way.
- **Watchful Eye Needed:** However, this path might have missed the intricacies of complex relationships among variables.

Wrapper Method (RFE - Recursive Feature Elimination):

- **Model-Guided Journey:** This approach navigated through predictors, seeking those contributing most to model performance.
- **Selected Enclave:** It spotlighted a smaller group of features ('width', 'bore', 'compression', 'city_mpg', 'fuel_type_gas').
- **Potential Limitation:** Yet, the journey might have bypassed valuable insights outside this selected group.

Embedded Methods (Lasso:

- **Regularization Route:** These methods used the power of regularization, consistently pointing toward the same set of features ('wheel_base', 'length', 'width', 'heights', 'curb_weight', 'engine_size', 'bore', 'stroke', 'compression', 'horse_power', 'peak_rpm', 'city_mpg', 'highway_mpg', 'fuel_type_gas') across different strengths.
- **Stable Path:** However, while stable, this path might have overlooked nuances in feature importance.

Full Linear Regression Model (Model1):

- **Coefficients as Clues:** This path revealed insights into individual variable impacts with coefficients, but warnings about multicollinearity and non-numeric values in 'fuel_type_gas' cast shadows on reliability.
- **Insights with Caveats:** It provided clarity but also raised concerns about variable interactions and model stability.

Comparing Paths:

- **Varied Approaches:** Each path brought unique strengths but also limitations in selecting impactful features.
- **Precision vs. Scope:** Some paths highlighted specific variables, while others aimed for a broader view.

- **Insights vs. Blind Spots:** Coefficients offer detailed insights, yet concerns about reliability remain.

Conclusion:

- **Diverse Terrain:** Each method offers a different landscape in selecting features.
- **Strengths and Caveats:** While some paths focus on specific attributes, others offer a broader perspective, each with its own limitations.
- **Harmonizing Insights:** Combining findings from these diverse paths may offer a more comprehensive understanding, crucial for robust feature selection in predicting automobile prices.

In this grand exploration, combining the strengths of different paths might unveil the most influential predictors, guiding us toward a better understanding of automobile price prediction in our dataset.

2.5 Bonus question (*extra 5 points*)



```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Standardizing the data
scaler = StandardScaler()
scaled_df2 = scaler.fit_transform(df2)

# Applying PCA
pca = PCA()
df2_pca = pca.fit_transform(scaled_df2)

ex_var_ratio = pca.explained_variance_ratio_
cum_var_ratio = np.cumsum(ex_var_ratio)
num_components = np.argmax(cum_var_ratio >= 0.95) + 1
```



1. Standardizing the data:

`scaler = StandardScaler()`: Initializes a `StandardScaler` object, which is used for standardizing the data.

`scaled_df2 = scaler.fit_transform(df2)`: Applies standardization to the original `DataFrame` `df2`, resulting in `scaled_df2`. Standardization involves scaling features to have a mean of 0 and a standard deviation of 1.

2. Applying PCA (Principal Component Analysis):

`pca = PCA()`: Initializes a PCA object.

`df2_pca = pca.fit_transform(scaled_df2)`: Applies PCA to the standardized data, resulting in a new DataFrame `df2_pca` where each column represents a principal component.

3.Explained variance ratio and cumulative variance ratio:

`ex_var_ratio = pca.explained_variance_ratio_`: Retrieves the explained variance ratio for each principal component. This ratio represents the proportion of the dataset's variance explained by each principal component.

`cum_var_ratio = np.cumsum(ex_var_ratio)`: Computes the cumulative sum of the explained variance ratios. `cum_var_ratio` represents the cumulative explained variance as more principal components are included.

4.Determining the number of components needed to explain at least 95% of the variance:

`num_components = np.argmax(cum_var_ratio >= 0.95) + 1`: Finds the index of the first element in `cum_var_ratio` that is greater than or equal to 0.95. The + 1 is added to account for zero-based indexing. This provides the number of principal components needed to explain at least 95% of the variance in the data.

The code performs data standardization, applies PCA to reduce dimensionality, and then determines the number of principal components needed to explain a specified percentage (95%) of the variance in the data. This information is crucial for

understanding the trade-off between dimensionality reduction and preserving information in the dataset.



```
[ ] from sklearn.model_selection import train_test_split
    from sklearn.metrics import r2_score

X_train, X_test, y_train, y_test = train_test_split(df2_pca[:, :num_components], df2['price'], test_size=0.2, random_state=41)
reg_pca = LinearRegression()
reg_pca.fit(X_train, y_train)
y_pred = reg_pca.predict(X_test)
r2_pca = r2_score(y_test, y_pred)
print(f"R-squared with {num_components} PCA components: {r2_pca}")
```

★ Output:-

R-squared with 8 PCA components: 0.9034013797858127

★ 1.Splitting the data into training and testing sets:

`train_test_split(df2_pca[:, :num_components], df2['price'], test_size=0.2, random_state=41)`: Splits the data into training and testing sets. The predictor variables (X) include the first `num_components` principal components, and the target variable (y) is the 'price' column from the original DataFrame `df2`. The `test_size=0.2` parameter specifies that 20% of the data will be used for testing, and `random_state=41` ensures reproducibility.

2.Creating a linear regression model:

`reg_pca = LinearRegression()`: Initializes a linear regression model using scikit-learn's `LinearRegression` class.

3.Fitting the model on the training data:

`reg_pca.fit(X_train, y_train)`: Fits the linear regression model to the training data, where `X_train` is the training set of principal components, and `y_train` is the corresponding target variable.

4.Making predictions on the test data:

`y_pred = reg_pca.predict(X_test)`: Uses the trained model to make predictions on the test set of principal components (`X_test`).

5.Calculating R-squared:

`r2_pca = r2_score(y_test, y_pred)`: Calculates the R-squared value, a metric that measures the proportion of the variance in the target variable ('price') that is explained by the model's predictions.

6.Printing the R-squared value:

`print(f"R-squared with {num_components} PCA components: {r2_pca}")`: Prints the R-squared value along with the number of principal components used in the model.

This code assesses the performance of a linear regression model with a reduced number of principal components on the test data, providing insights into how well the model captures the variation in the target variable.



```
▶ pc_loadings = pca.components_[0:8]
  comp_feature_df = pd.DataFrame(pc_loadings, columns=df2.columns)
  print(comp_feature_df)
```

★ Output:-

```
▶
```

	wheel_base	length	width	heights	curb_weight	engine_size	\
0	0.290047	0.329242	0.324857	0.113327	0.352442	0.321280	
1	0.205101	0.106136	0.097530	0.269293	0.051227	-0.055221	
2	0.229445	0.178026	-0.007540	0.579269	-0.038693	-0.256331	
3	-0.257018	-0.145007	-0.107423	-0.354444	0.005923	0.142842	
4	-0.064503	0.000850	0.062484	0.072632	0.026038	-0.044901	
5	0.114009	0.005035	0.179351	-0.069932	0.053998	0.256503	
6	-0.212875	-0.204405	-0.379356	0.573435	-0.039832	0.311784	
7	-0.324208	-0.104034	-0.327071	0.200524	0.093997	-0.117506	

	bore	stroke	compression	horse_power	peak_rpm	city_mpg	\
0	0.258763	0.054424	0.022413	0.295146	-0.085280	-0.305684	
1	-0.018457	0.133350	0.522074	-0.237651	-0.361603	0.250687	
2	0.133695	-0.563640	-0.206042	-0.226212	-0.071874	-0.043831	
3	0.395032	-0.611162	0.138460	0.105754	-0.406339	0.044639	
4	-0.234711	-0.438033	0.343483	0.142506	0.678171	0.007413	
5	-0.747518	-0.240980	-0.137120	-0.039024	-0.342971	0.127499	
6	0.110663	0.145549	-0.057528	0.225023	-0.022346	0.184382	
7	-0.341461	0.006623	0.132812	0.100668	-0.253240	-0.467865	

	highway_mpg	price	fuel_type_gas
0	-0.316334	0.318148	-0.046116
1	0.207876	-0.042560	-0.522664
2	-0.056973	-0.165215	0.214549
3	0.050952	0.113077	-0.113551
4	0.005470	0.248287	-0.277608
5	0.095249	0.260539	0.182188
6	0.220348	0.402392	0.094991
7	-0.468088	-0.158580	-0.206259

★ 1.Extracting the loadings of the first 8 principal components:

`pc_loadings = pca.components_[:8]`: The `pca.components_` attribute contains the loadings of each principal component. This code extracts the loadings of the first 8 principal components and stores them in the `pc_loadings` variable.

2.Creating a DataFrame to represent the relationship:

`comp_feature_df = pd.DataFrame(pc_loadings, columns=df2.columns)`: Creates a DataFrame `comp_feature_df` using the extracted loadings. Each row of the DataFrame corresponds to a principal component, and each column corresponds to an original feature from the DataFrame `df2`. This DataFrame represents the relationship between the original features and the principal components.

3.Printing the resulting DataFrame:

`print(comp_feature_df)`: Prints the DataFrame that shows the loadings of each original feature on each of the first 8 principal components.

Each element in the DataFrame indicates the weight or contribution of a specific original feature to a particular principal component. Positive or negative values signify the direction and strength of the relationship. This kind of DataFrame is often useful for interpreting the meaning of the principal components in terms of the original features.

Keep in mind that interpretation of loadings can be complex and may depend on the specific context of the data and the analysis. In general, higher absolute values of loadings indicate a stronger relationship between a feature and a principal component.



```
from sklearn.decomposition import PCA

# Fit PCA to your training data
pca = PCA()
pca.fit(X_train)

# Calculate the cumulative explained variance
cumulative_explained_variance = np.cumsum(pca.explained_variance_ratio_)

# Create a plot to visualize the cumulative explained variance
plt.figure(figsize=(8, 6))
plt.plot(cumulative_explained_variance, marker='o')
plt.xlabel("Number of Components")
plt.ylabel("Cumulative Explained Variance")
plt.title("Cumulative Explained Variance by Principal Components")
plt.grid()
plt.show()
```



Output:- 1. Fit PCA to your training data:

`pca = PCA()`: Initializes a PCA object.

`pca.fit(X_train)`: Fits the PCA model to the training data (`X_train`), where each row represents an observation and each column represents a principal component.

2. Calculate the cumulative explained variance:

`cumulative_explained_variance` = `np.cumsum(pca.explained_variance_ratio_)`: Calculates the cumulative sum of the explained variance ratios for each principal component. This array represents the cumulative amount of variance explained as more components are included.

3. Create a plot to visualize the cumulative explained variance:

`plt.figure(figsize=(8, 6))`: Specifies the size of the plot.

`plt.plot(cumulative_explained_variance, marker='o')`: Plots the cumulative explained variance with markers at each point.

`plt.xlabel("Number of Components")`: Sets the x-axis label.

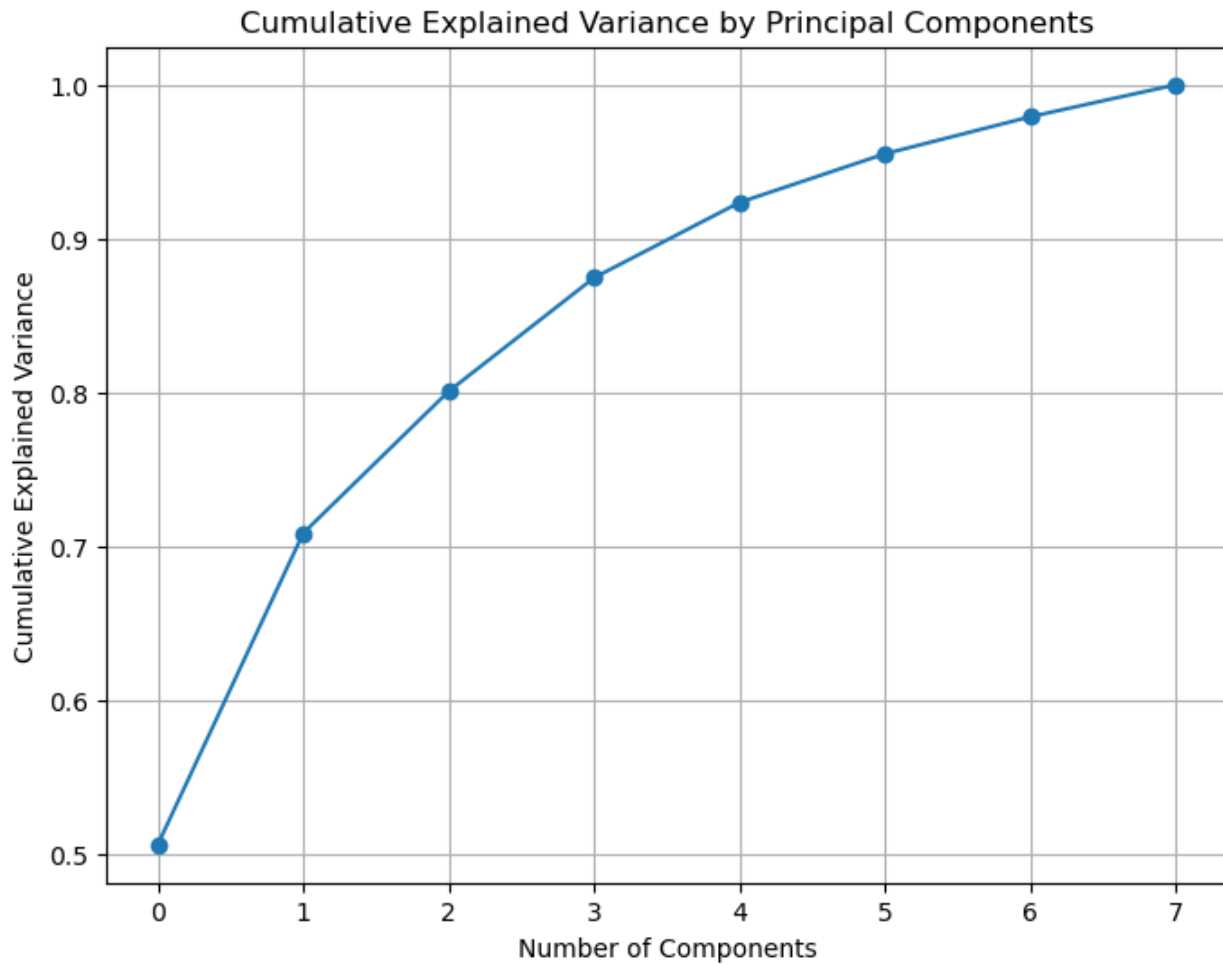
`plt.ylabel("Cumulative Explained Variance")`: Sets the y-axis label.

`plt.title("Cumulative Explained Variance by Principal Components")`: Sets the title of the plot.

`plt.grid()`: Adds a grid to the plot.

`plt.show()`: Displays the plot.


The resulting plot visually shows how much variance in the data is captured as the number of principal components increases, helping to inform decisions about the appropriate number of components to retain for dimensionality reduction.



```
▶ cumulative_variance = np.cumsum(np.round(pca.explained_variance_ratio_, decimals=4) * 100)
   explained_variance = list(zip(range(1, len(cumulative_variance) + 1), cumulative_variance))
   print(explained_variance)
```

★ Output:-

```
[(1, 50.63999999999999), (2, 70.86999999999999), (3, 80.08999999999999), (4, 87.46999999999998), (5, 92.35999999999999), (6, 95.51999999999998), (7, 97.92999999999998), (8, 100.00999999999998)]
```

 Component 1: Positive Contributions: 'length,' 'width,' 'curb_weight,' 'engine_size,' 'bore,' 'horse_power,' and 'price.' Minor Positive Contributions: 'wheel_base' and 'city_mpg.' Negative Contributions: 'heights,' 'peak_rpm,' 'highway_mpg,' and 'fuel_type_gas.'

Component 2: Dominated by 'heights' and 'price' with opposing contributions. 'length,' 'heights,' and 'city_mpg' also have discernible contributions.

Component 3: 'wheel_base' and 'length' make substantial contributions. 'heights' and 'bore' contribute but in opposing directions.

Component 4: Primarily influenced by 'heights' and 'bore' with conflicting contributions. 'engine_size' and 'stroke' also show notable contributions.

Component 5: 'curb_weight' and 'engine_size' hold significant contributions. 'stroke' and 'compression' make minor contributions.

Component 6: 'engine_size' and 'city_mpg' are the dominant contributors. 'bore' and 'highway_mpg' show minor contributions.

Component 7: 'bore' and 'stroke' have significant contributions but in opposite directions. 'heights' and 'highway_mpg' also exhibit noticeable contributions.

Component 8: 'heights' and 'compression' appear as primary contributors. 'wheel_base' and 'fuel_type_gas' have smaller, opposing contributions. These coefficients help to interpret how each original variable contributes to the creation of principal components, providing insights into their significance in reducing the dataset's dimensionality while preserving essential information.