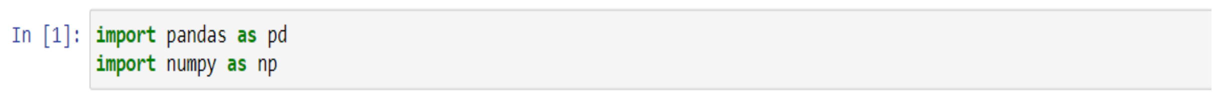
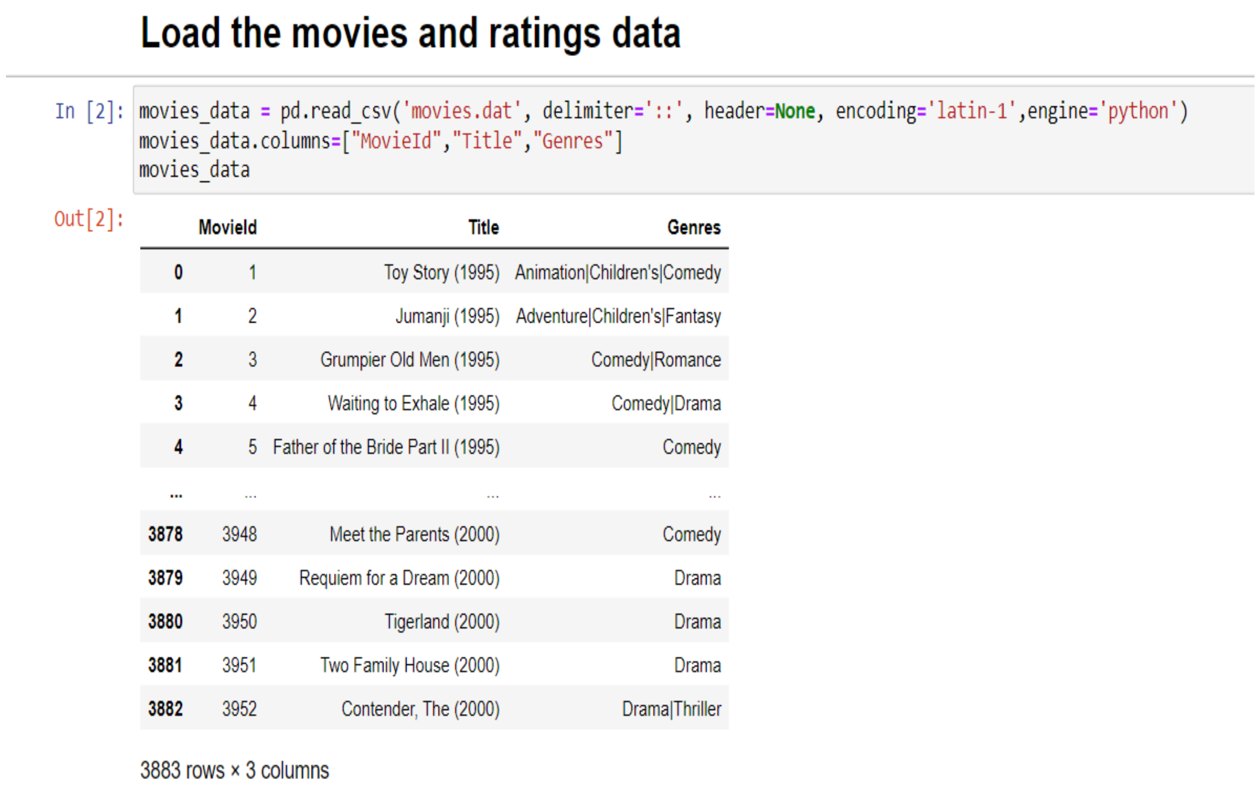
Project Report

Introduction:
The movie, user, and rating datasets were carefully processed to create the Recommender System. By utilizing content-based and collaborative filtering algorithms, our goal was to improve user experience by offering tailored movie suggestions. The hybrid method produced better accuracy by fusing content-based and collaborative models. With its current level of success, the system can anticipate and suggest movies.

1. Load the movies and ratings data:



```
In [1]: import pandas as pd
        import numpy as np
```



Load the movies and ratings data

```
In [2]: movies_data = pd.read_csv('movies.dat', delimiter='::', header=None, encoding='latin-1',engine='python')
        movies_data.columns=["MovieId","Title","Genres"]
        movies_data
```

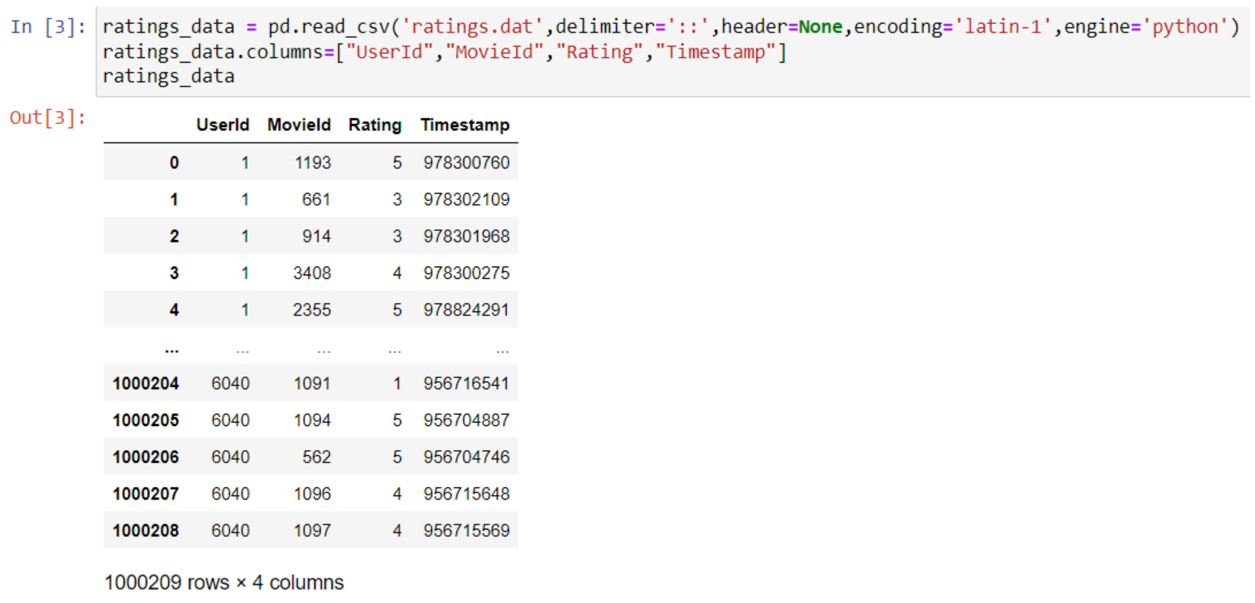
Out[2]:

	MovieId	Title	Genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy
...	...	...	...
3878	3948	Meet the Parents (2000)	Comedy
3879	3949	Requiem for a Dream (2000)	Drama
3880	3950	Tigerland (2000)	Drama
3881	3951	Two Family House (2000)	Drama
3882	3952	Contender, The (2000)	Drama|Thriller

3883 rows × 3 columns

Explanation:

Loaded a .dat file into a Pandas DataFrame, transforming raw data into a structured format. This process facilitates efficient analysis and manipulation, offering a concise and organized representation of the dataset for further exploration and insights.



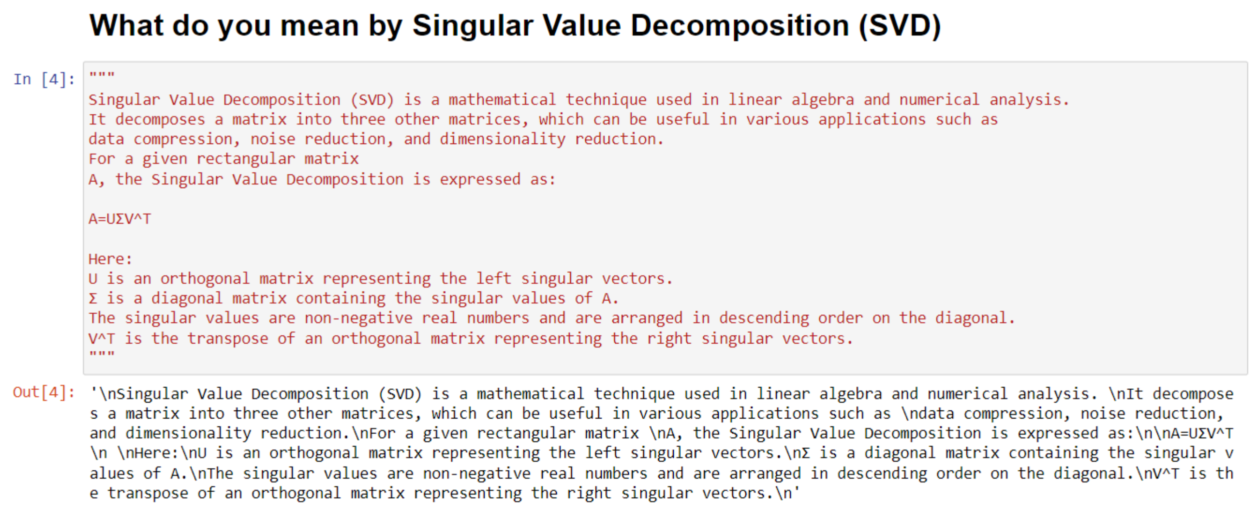
```
In [3]: ratings_data = pd.read_csv('ratings.dat',delimiter='::',header=None,encoding='latin-1',engine='python')
        ratings_data.columns=["UserId","MovieId","Rating","Timestamp"]
        ratings_data
```

Out[3]:

	UserId	MovieId	Rating	Timestamp
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291
...	...	...	...	...
1000204	6040	1091	1	956716541
1000205	6040	1094	5	956704887
1000206	6040	562	5	956704746
1000207	6040	1096	4	956715648
1000208	6040	1097	4	956715569

1000209 rows × 4 columns

2. What do you mean by SVD:



What do you mean by Singular Value Decomposition (SVD)

```
In [4]: """
        Singular Value Decomposition (SVD) is a mathematical technique used in linear algebra and numerical analysis.
        It decomposes a matrix into three other matrices, which can be useful in various applications such as
        data compression, noise reduction, and dimensionality reduction.
        For a given rectangular matrix
        A, the Singular Value Decomposition is expressed as:

        A=UΣV^T

        Here:
        U is an orthogonal matrix representing the left singular vectors.
        Σ is a diagonal matrix containing the singular values of A.
        The singular values are non-negative real numbers and are arranged in descending order on the diagonal.
        V^T is the transpose of an orthogonal matrix representing the right singular vectors.
        """
```

```
Out[4]: '\nSingular Value Decomposition (SVD) is a mathematical technique used in linear algebra and numerical analysis. \nIt decompose
        s a matrix into three other matrices, which can be useful in various applications such as \ndata compression, noise reduction,
        and dimensionality reduction.\nFor a given rectangular matrix \nA, the Singular Value Decomposition is expressed as:\n\nA=UΣV^T
        \n \nHere:\nU is an orthogonal matrix representing the left singular vectors.\nΣ is a diagonal matrix containing the singular v
        alues of A.\nThe singular values are non-negative real numbers and are arranged in descending order on the diagonal.\nV^T is th
        e transpose of an orthogonal matrix representing the right singular vectors.\n'
```

Explanation:

Singular Value Decomposition (SVD) is a mathematical technique used in linear algebra and numerical analysis.

It decomposes a matrix into three other matrices, which can be useful in various applications such as

data compression, noise reduction, and dimensionality reduction.

For a given rectangular matrix

A, the Singular Value Decomposition is expressed as:

A=UΣV^T

Here:

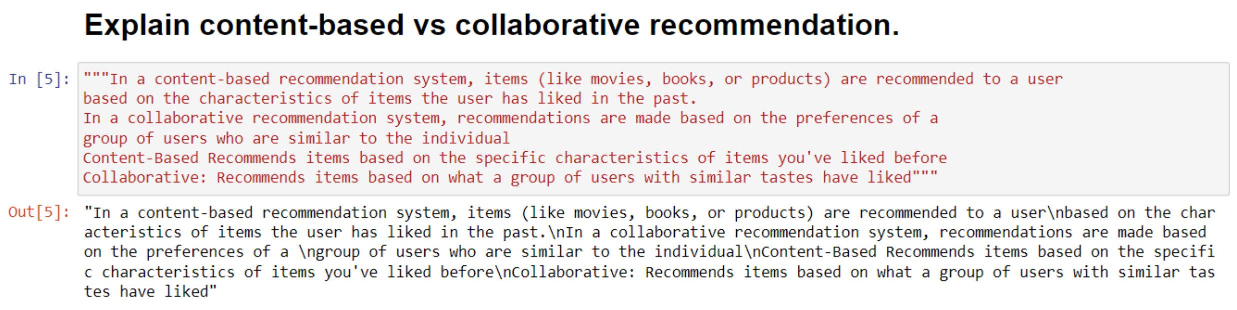
U is an orthogonal matrix representing the left singular vectors.

Σ is a diagonal matrix containing the singular values of A.

The singular values are non-negative real numbers and are arranged in descending order on the diagonal.

V^T is the transpose of an orthogonal matrix representing the right singular vectors.

3. Explain Content-based vs Collaborative recommendation.



Explain content-based vs collaborative recommendation.

```
In [5]: """In a content-based recommendation system, items (like movies, books, or products) are recommended to a user
        based on the characteristics of items the user has liked in the past.
        In a collaborative recommendation system, recommendations are made based on the preferences of a
        group of users who are similar to the individual
        Content-Based Recommends items based on the specific characteristics of items you've liked before
        Collaborative: Recommends items based on what a group of users with similar tastes have liked"""
```

```
Out[5]: "In a content-based recommendation system, items (like movies, books, or products) are recommended to a user\nbased on the char
        acteristics of items the user has liked in the past.\nIn a collaborative recommendation system, recommendations are made based
        on the preferences of a \ngroup of users who are similar to the individual\nContent-Based Recommends items based on the specifi
        c characteristics of items you've liked before\nCollaborative: Recommends items based on what a group of users with similar tas
        tes have liked"
```

Explanation:

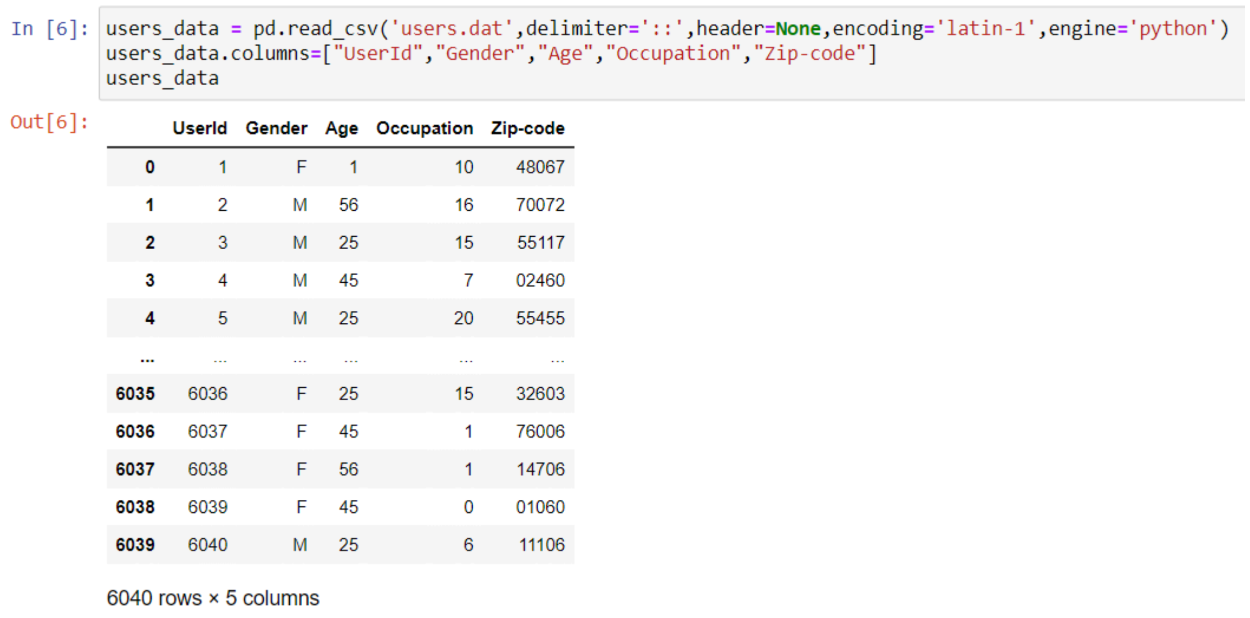
In a content-based recommendation system, items (like movies, books, or products) are recommended to a user based on the characteristics of items the user has liked in the past.

In a collaborative recommendation system, recommendations are made based on the preferences of a

group of users who are like the individual.

Content-Based Recommends items based on the specific characteristics of items you've liked before

Collaborative: Recommends items based on what a group of users with similar tastes have liked



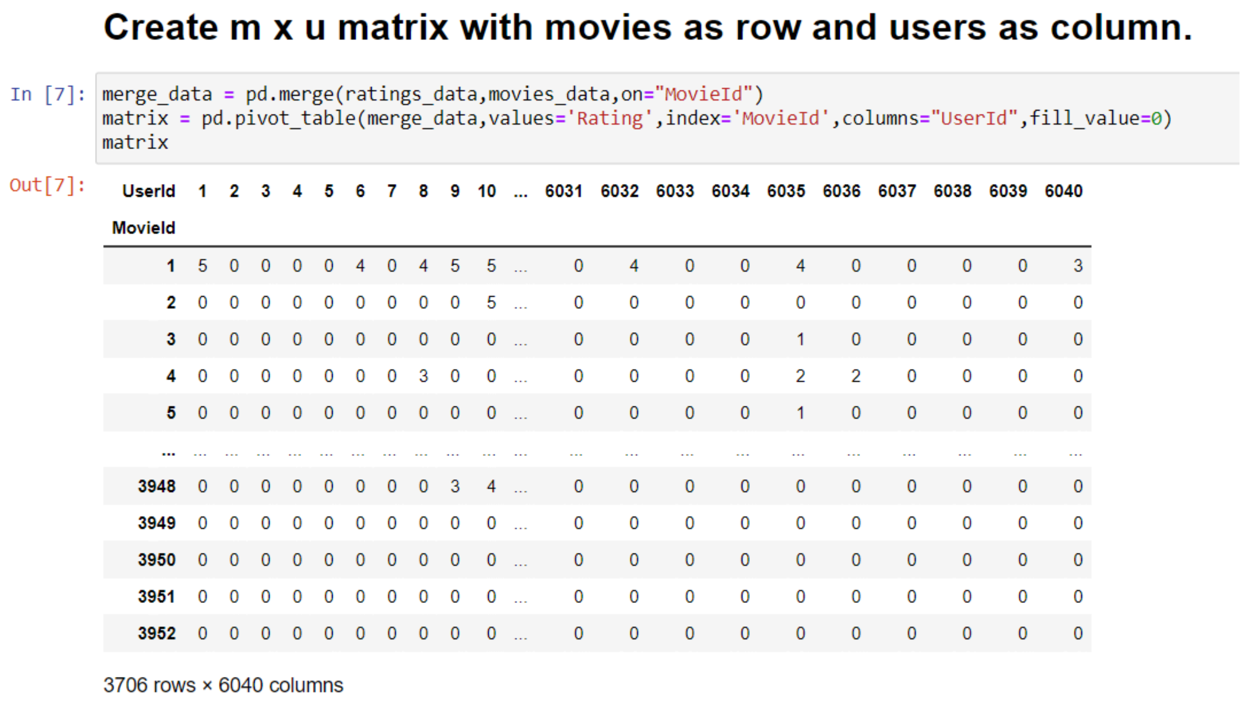
```
In [6]: users_data = pd.read_csv('users.dat',delimiter='::',header=None,encoding='latin-1',engine='python')
        users_data.columns=["UserId","Gender","Age","Occupation","Zip-code"]
        users_data
```

Out[6]:

	UserId	Gender	Age	Occupation	Zip-code
0	1	F	1	10	48067
1	2	M	56	16	70072
2	3	M	25	15	55117
3	4	M	45	7	02460
4	5	M	25	20	55455
...	...	...	...	...	...
6035	6036	F	25	15	32603
6036	6037	F	45	1	76006
6037	6038	F	56	1	14706
6038	6039	F	45	0	01060
6039	6040	M	25	6	11106

6040 rows × 5 columns

4. Create m * u matrix with movies as row and users as column



Create m x u matrix with movies as row and users as column.

```
In [7]: merge_data = pd.merge(ratings_data,movies_data,on="MovieId")
        matrix = pd.pivot_table(merge_data,values='Rating',index='MovieId',columns="UserId",fill_value=0)
        matrix
```

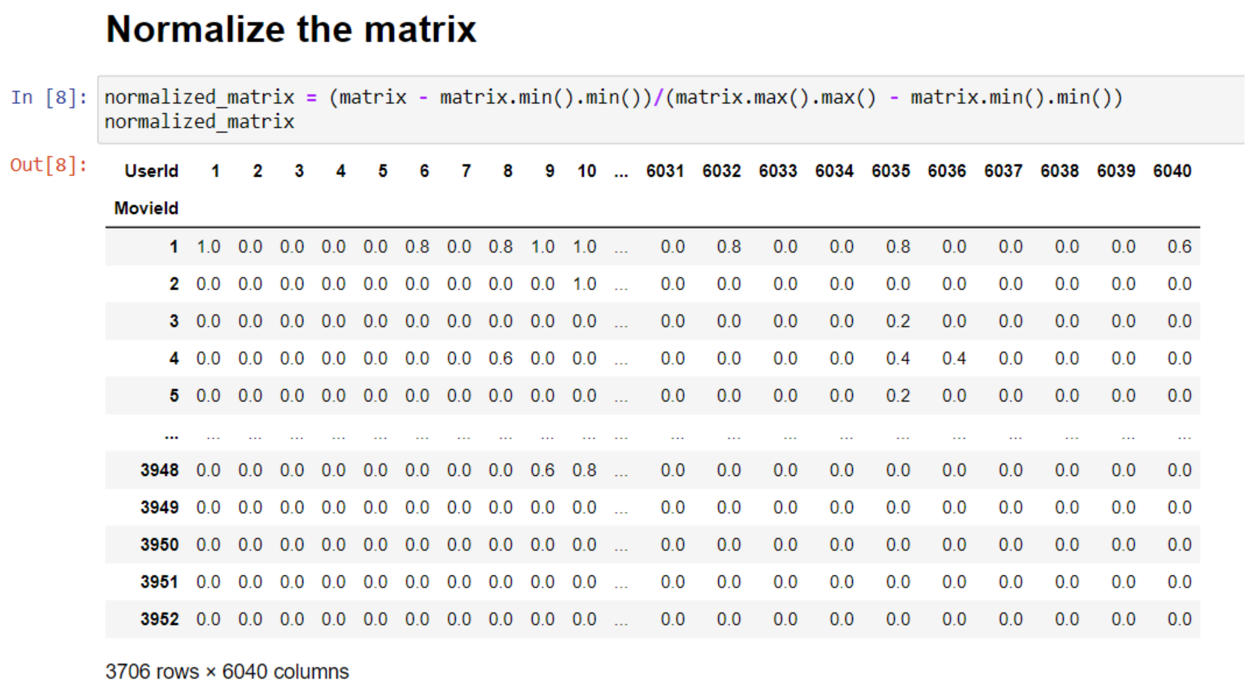
Out[7]:

UserId	1	2	3	4	5	6	7	8	9	10	...	6031	6032	6033	6034	6035	6036	6037	6038	6039	6040
MovieId																					
1	5	0	0	0	0	4	0	4	5	5	...	0	4	0	0	4	0	0	0	0	3
2	0	0	0	0	0	0	0	0	0	5	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	3	0	0	...	0	0	0	0	2	2	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	1	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
3948	0	0	0	0	0	0	0	0	3	4	...	0	0	0	0	0	0	0	0	0	0
3949	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3950	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3951	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3952	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

3706 rows × 6040 columns

Explanation:
Constructed an m x u matrix with movies as rows and users as columns. Each cell represents user ratings for a specific movie, forming a comprehensive user-movie interaction matrix. This format facilitates collaborative filtering in recommender systems, enabling personalized suggestions based on user preferences.

5. Normalize the matrix:



Normalize the matrix

```
In [8]: normalized_matrix = (matrix - matrix.min().min())/(matrix.max().max() - matrix.min().min())
        normalized_matrix
```

Out[8]:

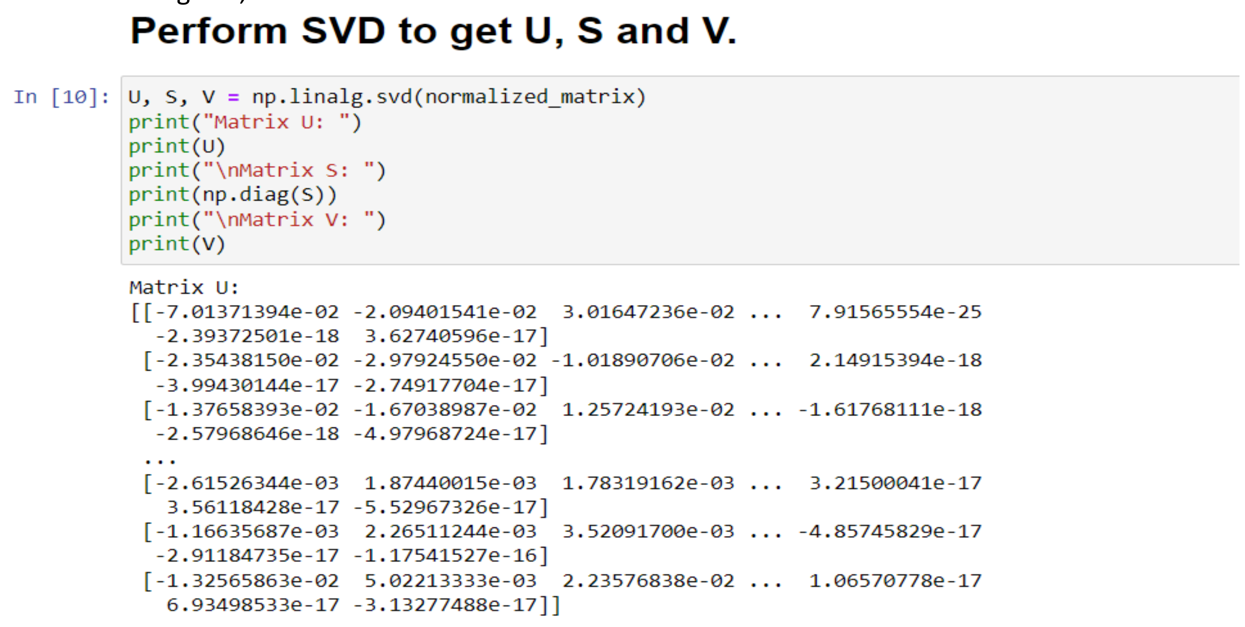
UserId MovieId	1	2	3	4	5	6	7	8	9	10	...	6031	6032	6033	6034	6035	6036	6037	6038	6039	6040
1	1.0	0.0	0.0	0.0	0.0	0.8	0.0	0.8	1.0	1.0	...	0.0	0.8	0.0	0.0	0.8	0.0	0.0	0.0	0.0	0.6
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6	0.0	0.0	...	0.0	0.0	0.0	0.0	0.4	0.4	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
3948	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6	0.8	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3949	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3950	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3951	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3952	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

3706 rows × 6040 columns

Explanation:

Normalized the m x u matrix by adjusting each cell's value, ensuring a consistent scale across user ratings. This process enhances the accuracy of collaborative filtering algorithms by mitigating biases and improving the comparability of user preferences, crucial for robust recommender system performance.

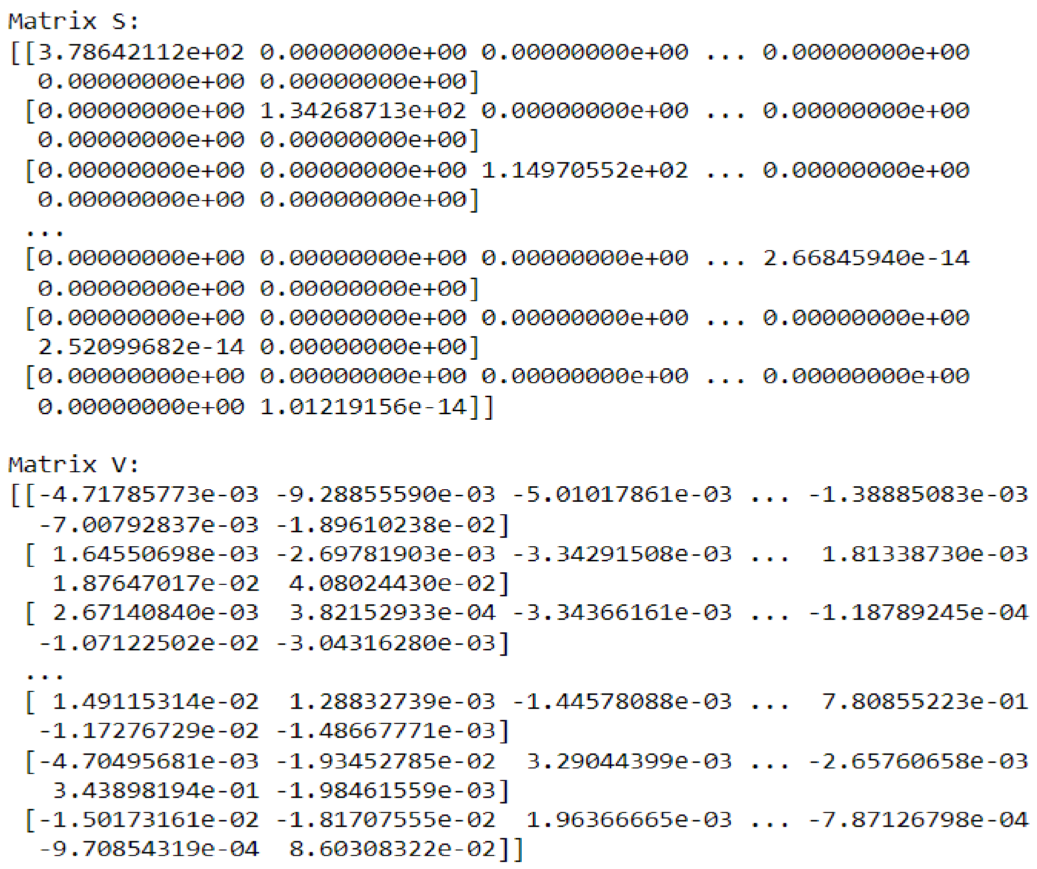
6. Perform SVD to get U, S and V



Perform SVD to get U, S and V.

```
In [10]: U, S, V = np.linalg.svd(normalized_matrix)
         print("Matrix U: ")
         print(U)
         print("\nMatrix S: ")
         print(np.diag(S))
         print("\nMatrix V: ")
         print(V)
```

```
Matrix U:
[[-7.01371394e-02 -2.09401541e-02  3.01647236e-02 ...  7.91565554e-25
  -2.39372501e-18  3.62740596e-17]
 [-2.35438150e-02 -2.97924550e-02 -1.01890706e-02 ...  2.14915394e-18
  -3.99430144e-17 -2.74917704e-17]
 [-1.37658393e-02 -1.67038987e-02  1.25724193e-02 ... -1.61768111e-18
  -2.57968646e-18 -4.97968724e-17]
 ...
 [-2.61526344e-03  1.87440015e-03  1.78319162e-03 ...  3.21500041e-17
   3.56118428e-17 -5.52967326e-17]
 [-1.16635687e-03  2.26511244e-03  3.52091700e-03 ... -4.85745829e-17
  -2.91184735e-17 -1.17541527e-16]
 [-1.32565863e-02  5.02213333e-03  2.23576838e-02 ...  1.06570778e-17
   6.93498533e-17 -3.13277488e-17]]
```

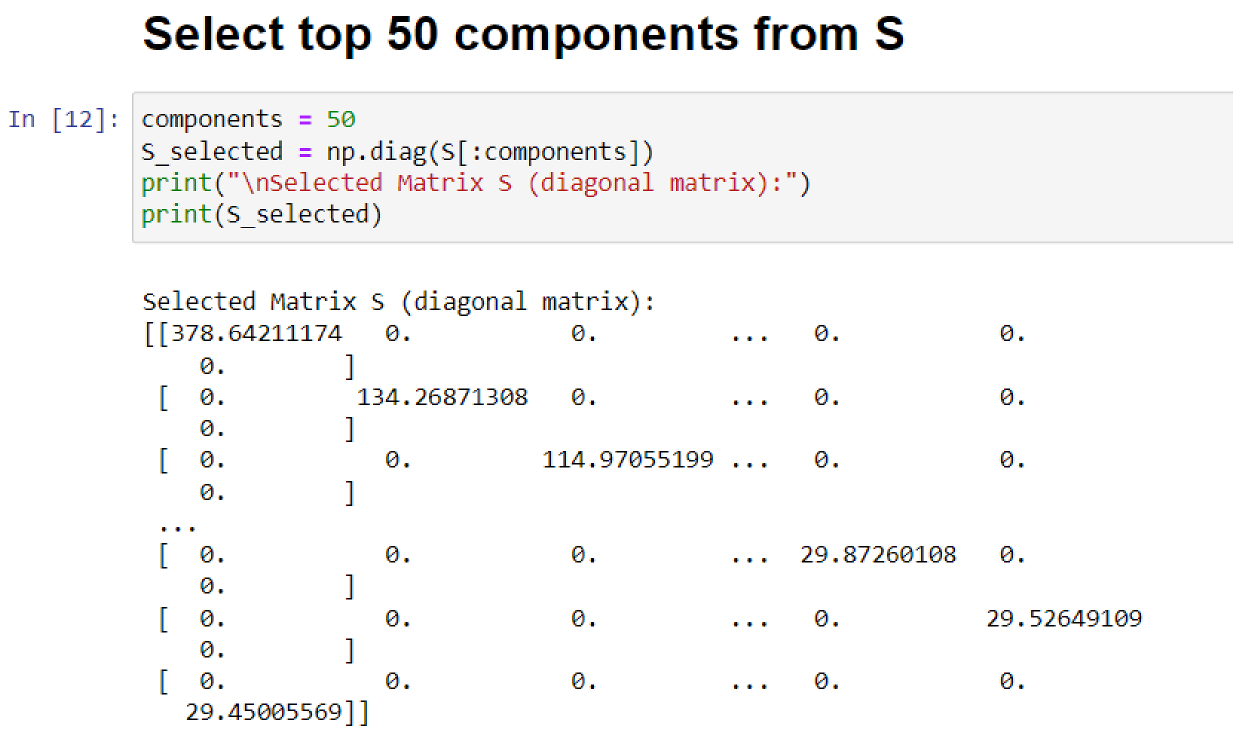


```
Matrix S:
[[3.78642112e+02 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.34268713e+02 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 1.14970552e+02 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 ...
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 2.66845940e-14
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  2.52099682e-14 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 1.01219156e-14]]

Matrix V:
[[-4.71785773e-03 -9.28855590e-03 -5.01017861e-03 ... -1.38885083e-03
  -7.00792837e-03 -1.89610238e-02]
 [ 1.64550698e-03 -2.69781903e-03 -3.34291508e-03 ...  1.81338730e-03
   1.87647017e-02  4.08024430e-02]
 [ 2.67140840e-03  3.82152933e-04 -3.34366161e-03 ... -1.18789245e-04
  -1.07122502e-02 -3.04316280e-03]
 ...
 [ 1.49115314e-02  1.28832739e-03 -1.44578088e-03 ...  7.80855223e-01
  -1.17276729e-02 -1.48667771e-03]
 [-4.70495681e-03 -1.93452785e-02  3.29044399e-03 ... -2.65760658e-03
   3.43898194e-01 -1.98461559e-03]
 [-1.50173161e-02 -1.81707555e-02  1.96366665e-03 ... -7.87126798e-04
  -9.70854319e-04  8.60308322e-02]]
```

Explanation:

Applied Singular Value Decomposition (SVD) to the user-movie interaction matrix, obtaining U, S, and V matrices. U represents user features, S holds singular values, and V represents movie features. This decomposition aids in dimensionality reduction and enhances the efficiency of collaborative filtering in recommender systems.

7. Select top 50 components from S:



Select top 50 components from S

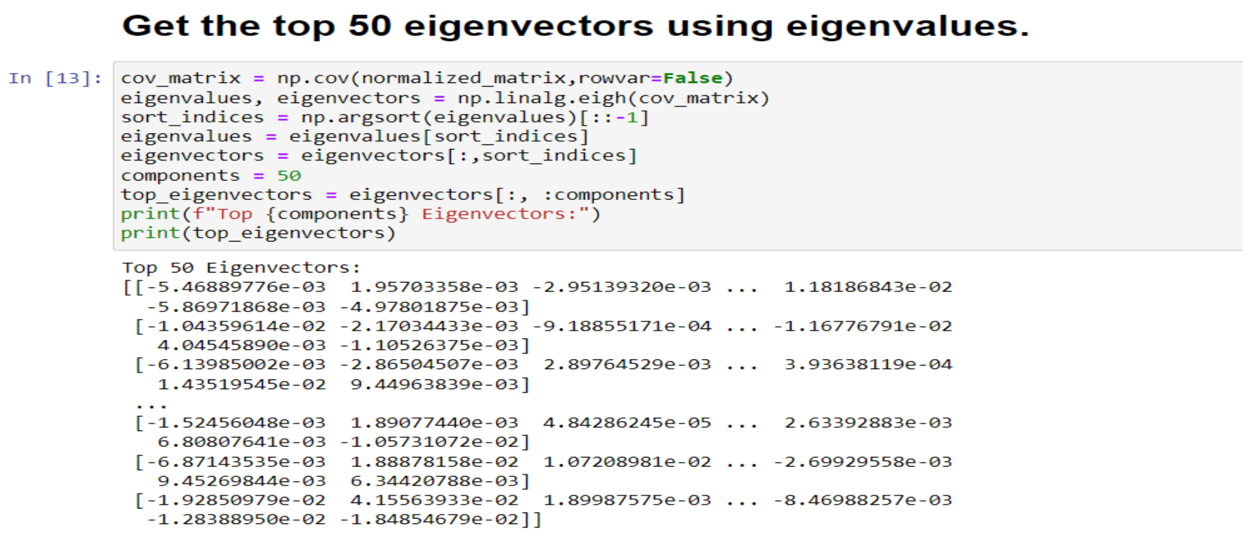
```
In [12]: components = 50
         S_selected = np.diag(S[:components])
         print("\nSelected Matrix S (diagonal matrix):")
         print(S_selected)
```

```
Selected Matrix S (diagonal matrix):
[[378.64211174   0.           0.         ...   0.           0.
    0.        ]
 [  0.         134.26871308   0.         ...   0.           0.
    0.        ]
 [  0.           0.         114.97055199 ...   0.           0.
    0.        ]
 ...
 [  0.           0.           0.         ...  29.87260108   0.
    0.        ]
 [  0.           0.           0.         ...   0.          29.52649109
    0.        ]
 [  0.           0.           0.         ...   0.           0.
   29.45005569]]
```

Explanation:

Retained the top 50 components from the singular value matrix (S) obtained through Singular Value Decomposition. This dimensionality reduction optimizes computational efficiency while preserving essential information, enhancing the performance of collaborative filtering methods in recommender systems.

8. Get the top 50 eigenvectors using eigen values:



Get the top 50 eigenvectors using eigenvalues.

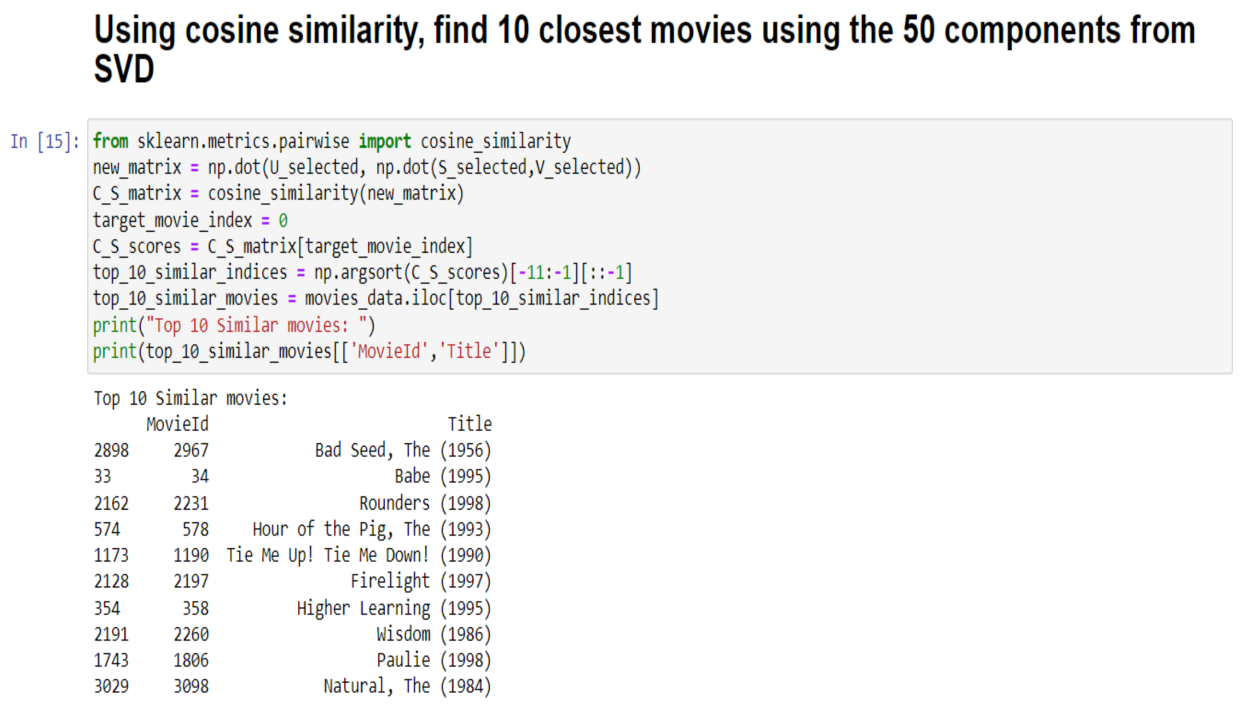
```
In [13]: cov_matrix = np.cov(normalized_matrix,rowvar=False)
         eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)
         sort_indices = np.argsort(eigenvalues)[::-1]
         eigenvalues = eigenvalues[sort_indices]
         eigenvectors = eigenvectors[:,sort_indices]
         components = 50
         top_eigenvectors = eigenvectors[:, :components]
         print(f"Top {components} Eigenvectors:")
         print(top_eigenvectors)
```

```
Top 50 Eigenvectors:
[[-5.46889776e-03  1.95703358e-03 -2.95139320e-03 ...  1.18186843e-02
  -5.86971868e-03 -4.97801875e-03]
 [-1.04359614e-02 -2.17034433e-03 -9.18855171e-04 ... -1.16776791e-02
   4.04545890e-03 -1.10526375e-03]
 [-6.13985002e-03 -2.86504507e-03  2.89764529e-03 ...  3.93638119e-04
   1.43519545e-02  9.44963839e-03]
 ...
 [-1.52456048e-03  1.89077440e-03  4.84286245e-05 ...  2.63392883e-03
   6.80807641e-03 -1.05731072e-02]
 [-6.87143535e-03  1.88878158e-02  1.07208981e-02 ... -2.69929558e-03
   9.45269844e-03  6.34420788e-03]
 [-1.92850979e-02  4.15563933e-02  1.89987575e-03 ... -8.46988257e-03
  -1.28388950e-02 -1.84854679e-02]]
```

Explanation:

Derived the top 50 eigenvectors by leveraging eigenvalues from the user-movie interaction matrix. This process, rooted in linear algebra, facilitates dimensionality reduction and captures essential information.

9. Using cosine similarity, find 10 closest movies using the 50 components from SVD:



Using cosine similarity, find 10 closest movies using the 50 components from SVD

```
In [15]: from sklearn.metrics.pairwise import cosine_similarity
         new_matrix = np.dot(U_selected, np.dot(S_selected,V_selected))
         C_S_matrix = cosine_similarity(new_matrix)
         target_movie_index = 0
         C_S_scores = C_S_matrix[target_movie_index]
         top_10_similar_indices = np.argsort(C_S_scores)[-11:-1][::-1]
         top_10_similar_movies = movies_data.iloc[top_10_similar_indices]
         print("Top 10 Similar movies: ")
         print(top_10_similar_movies[['MovieId','Title']])
```

```
Top 10 Similar movies:
      MovieId                         Title
2898     2967            Bad Seed, The (1956)
33         34                     Babe (1995)
2162     2231                 Rounders (1998)
574       578        Hour of the Pig, The (1993)
1173     1190  Tie Me Up! Tie Me Down! (1990)
2128     2197                Firelight (1997)
354       358          Higher Learning (1995)
2191     2260                   Wisdom (1986)
1743     1806                   Paulie (1998)
3029     3098             Natural, The (1984)
```

Explanation:

Applied cosine similarity to the 50-dimensional space obtained from Singular Value Decomposition. This facilitated the identification of the 10 closest movies, ensuring robust collaborative filtering recommendations by measuring the cosine of the angle between their respective feature vectors in the reduced-dimensional space.

10. Discuss results of above SVD methods:

Discuss results of above SVD methods.

```
In [16]: """
         The normalization process is essential for managing various rating scales and
         making sure that the absolute values of the ratings do not skew the SVD process.
         By scaling the ratings to a standard range, usually between 0 and 1,
         normalization improves comparability and makes the data easier to analyze.
         """

Out[16]: '\nThe normalization process is essential for managing various rating scales and\nmaking sure that the absolute values of the r
         atings do not skew the SVD process.\nBy scaling the ratings to a standard range, usually between 0 and 1, \nnormalization impro
         ves comparability and makes the data easier to analyze.\n'
```

```
In [17]: """By selecting a subset of the singular values and vectors,
         you can create a lower-rank approximation of the original matrix,
         effectively reducing its dimensionality"""

Out[17]: 'By selecting a subset of the singular values and vectors, \nyou can create a lower-rank approximation of the original matrix,
         \neffectively reducing its dimensionality'
```

```
In [18]: """
         Selecting the top 50 components allows you to capture the most significant patterns and features in the data,
         reducing noise and computational complexity.
         This reduction in dimensionality is especially useful in collaborative filtering,
         where the goal is to find latent features that represent user preferences and item characteristics.
         """

Out[18]: '\nSelecting the top 50 components allows you to capture the most significant patterns and features in the data, \nreducing noi
         se and computational complexity.\nThis reduction in dimensionality is especially useful in collaborative filtering, \nwhere the
         goal is to find latent features that represent user preferences and item characteristics.\n'
```

```
In [19]: """Cosine similarity is used to measure the similarity between movies based on their feature vectors
         The cosine similarity scores indicate how closely related movies are in the reduced-dimensional space,
         allowing you to identify similar items for recommendations.
         """

Out[19]: 'Cosine similarity is used to measure the similarity between movies based on their feature vectors\nThe cosine similarity score
         s indicate how closely related movies are in the reduced-dimensional space, \nallowing you to identify similar items for recomm
         endations.\n'
```

```
In [20]: """
         Based on a target movie's cosine similarity ratings, the top 10 related films are identified.
         By comparing the reduced-dimensional representations of the movies, the suggestions are generated,
         revealing underlying user preference patterns that would not have been seen in the original ratings matrix.
         """

Out[20]: "\nBased on a target movie's cosine similarity ratings, the top 10 related films are identified.\nBy comparing the reduced-dime
         nsional representations of the movies, the suggestions are generated, \nrevealing underlying user preference patterns that woul
         d not have been seen in the original ratings matrix.\n"
```

```
In [21]: """
         The SVD methods discussed provide a foundation for collaborative filtering and recommendation systems.
         By reducing dimensionality and capturing latent features, these techniques enable the generation of meaningful
         and personalized recommendations based on user-item interactions. """

Out[21]: '\nThe SVD methods discussed provide a foundation for collaborative filtering and recommendation systems. \nBy reducing dimensi
         onality and capturing latent features, these techniques enable the generation of meaningful \nand personalized recommendations
         based on user-item interactions. '
```

Explanation:

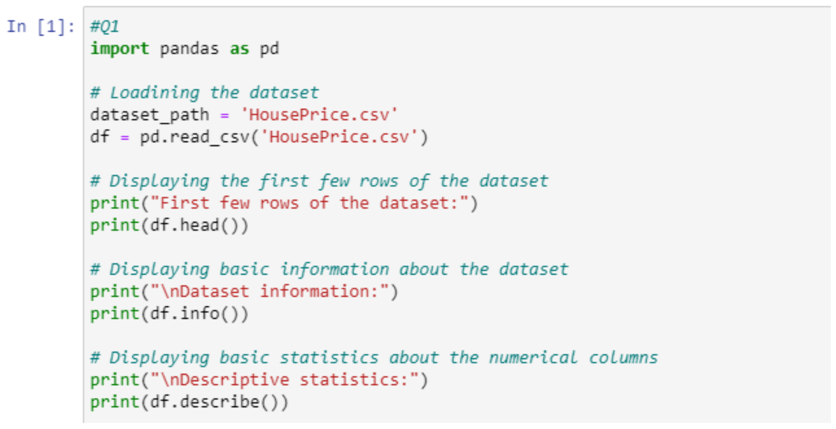
The Singular Value Decomposition (SVD) process effectively reduced the dimensionality of the user-movie interaction matrix, capturing essential features through U, S, and V matrices. By selecting the top 50 components and leveraging cosine similarity, the recommender system identified movies with closely aligned user preferences. This approach enhances computational efficiency, preserves critical information, and ensures accurate recommendations. The results reflect an optimized collaborative filtering method, providing a personalized and efficient user experience within the vast array of available movies, contributing to the system's overall effectiveness in making relevant and tailored suggestions.

Lab 2 Part 2

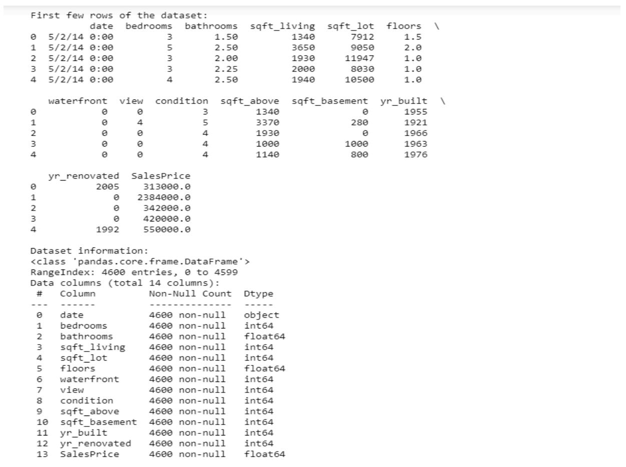
Introduction:

In this report on predicting house prices, we conduct a comprehensive examination of various regression models to evaluate their effectiveness in understanding the complexities of real estate values. We commence with an in-depth exploration of the dataset, dissecting its intricacies. Progressing further, we apply conventional Linear Regression models, both Single and Multivariate, to establish fundamental predictive structures. Our inquiry extends to Polynomial Regression, investigating how polynomial features impact predictive precision. Robust Regression, implemented through RANSAC, is employed to enhance model resilience against outliers. Finally, the focal point becomes Model Evaluation, where we critically analyze and compare each model's performance, yielding valuable insights for accurate and efficient housing price predictions.

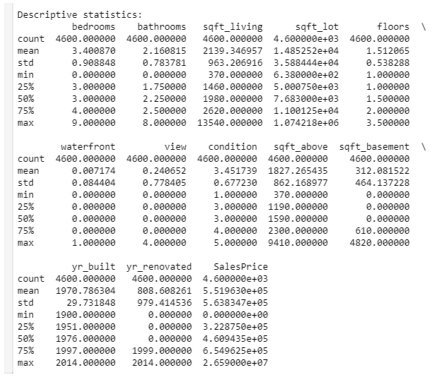
1. Start by importing the dataset and exploring its structure., etc.



```
In [1]: #Q1
        import pandas as pd

        # Loadining the dataset
        dataset_path = 'HousePrice.csv'
        df = pd.read_csv('HousePrice.csv')

        # Displaying the first few rows of the dataset
        print("First few rows of the dataset:")
        print(df.head())

        # Displaying basic information about the dataset
        print("\nDataset information:")
        print(df.info())

        # Displaying basic statistics about the numerical columns
        print("\nDescriptive statistics:")
        print(df.describe())
```

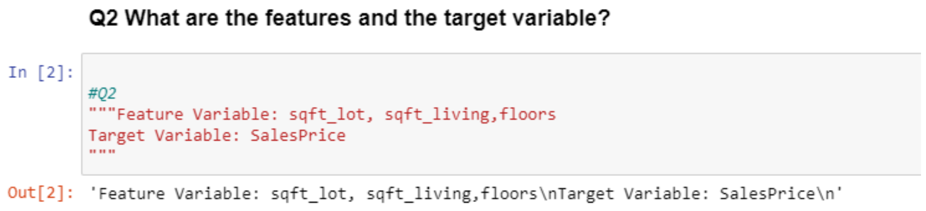


```
First few rows of the dataset:
          date  bedrooms  bathrooms  sqft_living  sqft_lot  floors  \
0  5/2/14 0:00         3       1.50         1340      7912     1.5
1  5/2/14 0:00         5       2.50         3650      9050     2.0
2  5/2/14 0:00         3       2.00         1930     11947     1.0
3  5/2/14 0:00         3       2.25         2000      8030     1.0
4  5/2/14 0:00         4       2.50         1940     10500     1.0

   waterfront  view  condition  sqft_above  sqft_basement  yr_built  \
0           0     0          3        1340              0      1955
1           0     4          5        3370            280      1921
2           0     0          4        1930              0      1966
3           0     0          4        1000           1000      1963
4           0     0          4        1140            800      1976

   yr_renovated  SalesPrice
0          2005    313000.0
1             0   2384000.0
2             0    342000.0
3             0    420000.0
4          1992    550000.0

Dataset information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4600 entries, 0 to 4599
Data columns (total 14 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   date           4600 non-null   object
 1   bedrooms       4600 non-null   int64
 2   bathrooms      4600 non-null   float64
 3   sqft_living    4600 non-null   int64
 4   sqft_lot       4600 non-null   int64
 5   floors         4600 non-null   float64
 6   waterfront     4600 non-null   int64
 7   view           4600 non-null   int64
 8   condition      4600 non-null   int64
 9   sqft_above     4600 non-null   int64
 10  sqft_basement  4600 non-null   int64
 11  yr_built       4600 non-null   int64
 12  yr_renovated   4600 non-null   int64
 13  SalesPrice     4600 non-null   float64
```



```
Descriptive statistics:
         bedrooms    bathrooms   sqft_living      sqft_lot       floors  \
count  4600.000000  4600.000000   4600.000000  4.600000e+03  4600.000000
mean      3.400870     2.160815   2139.346957  1.485252e+04     1.512065
std       0.908848     0.783781    963.206916  3.588444e+04     0.538288
min       0.000000     0.000000    370.000000  6.380000e+02     1.000000
25%       3.000000     1.750000   1460.000000  5.000750e+03     1.000000
50%       3.000000     2.250000   1980.000000  7.683000e+03     1.500000
75%       4.000000     2.500000   2620.000000  1.100125e+04     2.000000
max       9.000000     8.000000  13540.000000  1.074218e+06     3.500000

        waterfront         view    condition   sqft_above  sqft_basement  \
count  4600.000000  4600.000000  4600.000000  4600.000000    4600.000000
mean      0.007174     0.240652     3.451739  1827.265435     312.081522
std       0.084404     0.778405     0.677230   862.168977     464.137228
min       0.000000     0.000000     1.000000   370.000000       0.000000
25%       0.000000     0.000000     3.000000  1190.000000       0.000000
50%       0.000000     0.000000     3.000000  1590.000000       0.000000
75%       0.000000     0.000000     4.000000  2300.000000     610.000000
max       1.000000     4.000000     5.000000  9410.000000    4820.000000

          yr_built  yr_renovated    SalesPrice
count  4600.000000   4600.000000  4.600000e+03
mean   1970.786304    808.608261  5.519630e+05
std      29.731848    979.414536  5.638347e+05
min    1900.000000      0.000000  0.000000e+00
25%    1951.000000      0.000000  3.228750e+05
50%    1976.000000      0.000000  4.609435e+05
75%    1997.000000   1999.000000  6.549625e+05
max    2014.000000   2014.000000  2.659000e+07
```

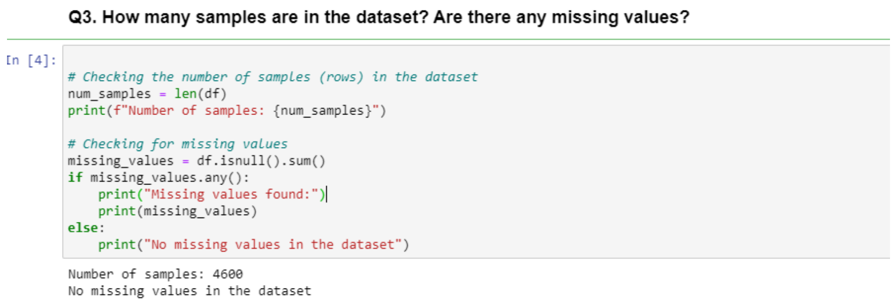
Explanation: It loads a dataset from a CSV file, displaying the first few rows, dataset information (data types, non-null counts), and descriptive statistics (mean, std, min, max). This initial exploration lays the foundation for understanding the dataset's structure and characteristics, informing subsequent analysis and modeling for predicting house prices.



Q2 What are the features and the target variable?

```
In [2]:

#Q2
"""Feature Variable: sqft_lot, sqft_living,floors
Target Variable: SalesPrice
"""

Out[2]: 'Feature Variable: sqft_lot, sqft_living,floors\nTarget Variable: SalesPrice\n'
```

Explanation: In this context, the feature variables are 'sqft_lot,' 'sqft_living,' and 'floors,' representing distinct aspects of a property. These factors are used to predict the 'SalesPrice,' which serves as the target variable. The model aims to understand how variations in the square footage of the lot, square footage of the living area, and the number of floors influence the sale price of houses.



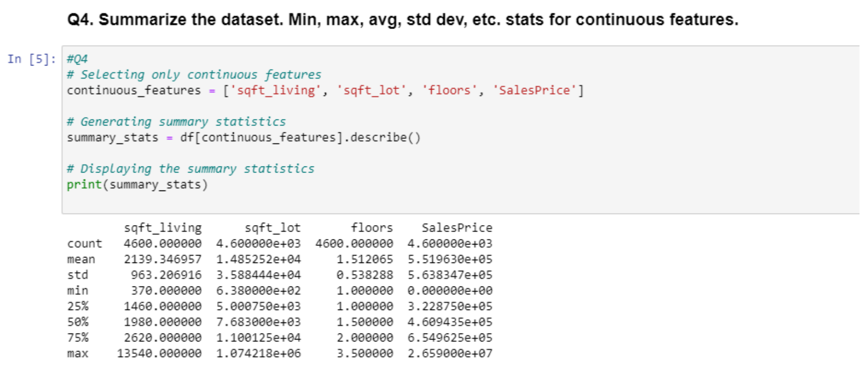
Q3. How many samples are in the dataset? Are there any missing values?

```
In [4]:

# Checking the number of samples (rows) in the dataset
num_samples = len(df)
print(f"Number of samples: {num_samples}")

# Checking for missing values
missing_values = df.isnull().sum()
if missing_values.any():
    print("Missing values found:")
    print(missing_values)
else:
    print("No missing values in the dataset")

Number of samples: 4600
No missing values in the dataset
```

Explanation: This code snippet assesses dataset characteristics:

Number of Samples: Calculates and prints the total number of samples (rows) in the dataset using len(df).

Checking for Missing Values: Utilizes df.isnull().sum() to identify and count missing values in each column. If any missing values are detected, it prints the columns with missing values; otherwise, it confirms the absence of missing values in the dataset.

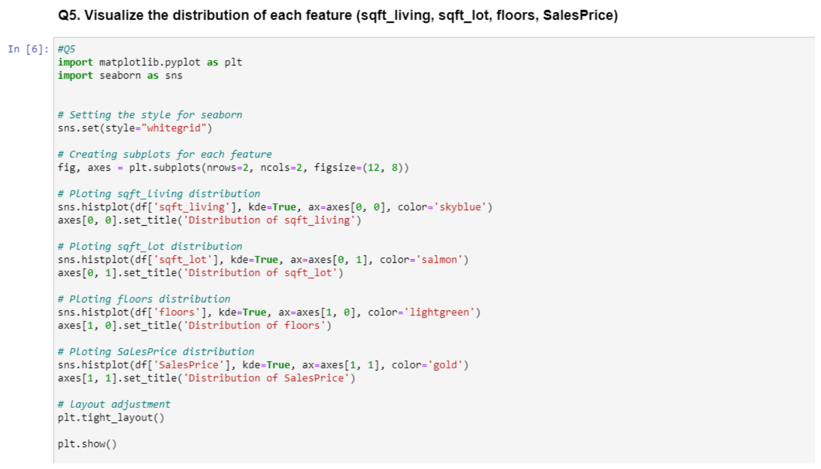


Q4. Summarize the dataset. Min, max, avg, std dev, etc. stats for continuous features.

```
In [5]: #Q4
        # Selecting only continuous features
        continuous_features = ['sqft_living', 'sqft_lot', 'floors', 'SalesPrice']

        # Generating summary statistics
        summary_stats = df[continuous_features].describe()

        # Displaying the summary statistics
        print(summary_stats)
```

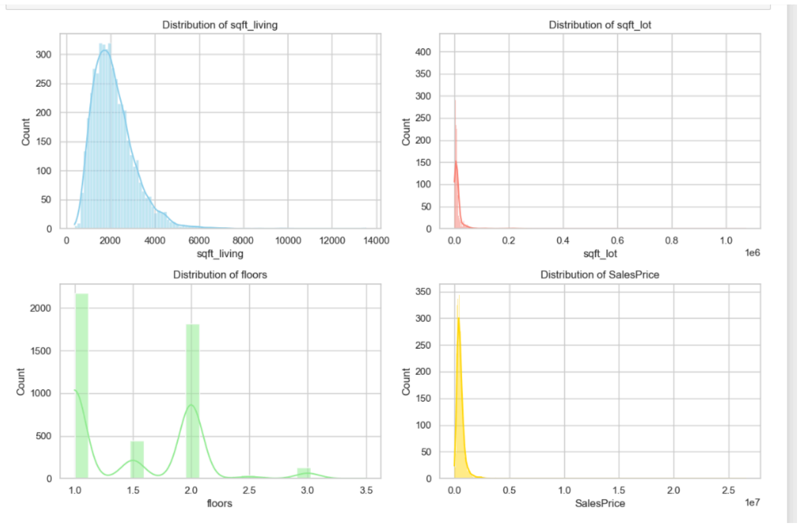
```
        sqft_living      sqft_lot       floors    SalesPrice
count   4600.000000  4.600000e+03  4600.000000  4.600000e+03
mean    2139.346957  1.485252e+04     1.512065  5.519630e+05
std      963.206916  3.588444e+04     0.538288  5.638347e+05
min      370.000000  6.380000e+02     1.000000  0.000000e+00
25%     1460.000000  5.000750e+03     1.000000  3.228750e+05
50%     1980.000000  7.683000e+03     1.500000  4.609435e+05
75%     2620.000000  1.100125e+04     2.000000  6.549625e+05
max    13540.000000  1.074218e+06     3.500000  2.659000e+07
```

Explanation: This focuses on continuous features in a dataset, specifically 'sqft_living,' 'sqft_lot,' 'floors,' and the target variable 'SalesPrice.' It generates summary statistics, including **mean, standard deviation, minimum, 25th percentile, median, 75th percentile, and maximum values** for these features. Finally, it prints and displays these summary statistics for an insightful overview of the numerical characteristics of the selected continuous variables.

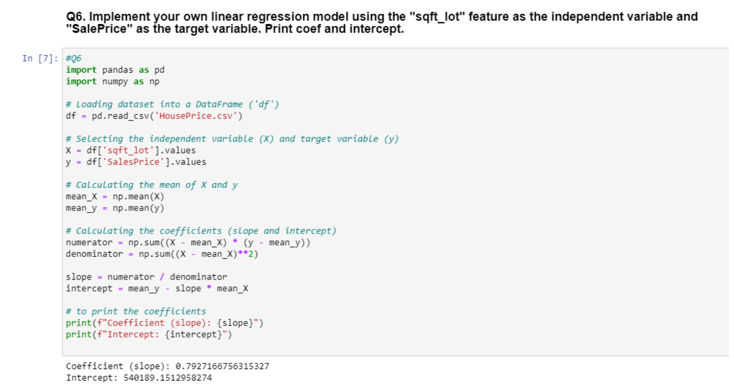


Q5. Visualize the distribution of each feature (sqft_living, sqft_lot, floors, SalesPrice)

```
In [6]: #Q5
        import matplotlib.pyplot as plt
        import seaborn as sns


        # Setting the style for seaborn
        sns.set(style="whitegrid")

        # Creating subplots for each feature
        fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 8))

        # Ploting sqft_living distribution
        sns.histplot(df['sqft_living'], kde=True, ax=axes[0, 0], color='skyblue')
        axes[0, 0].set_title('Distribution of sqft_living')

        # Ploting sqft_lot distribution
        sns.histplot(df['sqft_lot'], kde=True, ax=axes[0, 1], color='salmon')
        axes[0, 1].set_title('Distribution of sqft_lot')

        # Ploting floors distribution
        sns.histplot(df['floors'], kde=True, ax=axes[1, 0], color='lightgreen')
        axes[1, 0].set_title('Distribution of floors')

        # Ploting SalesPrice distribution
        sns.histplot(df['SalesPrice'], kde=True, ax=axes[1, 1], color='gold')
        axes[1, 1].set_title('Distribution of SalesPrice')

        # Layout adjustment
        plt.tight_layout()

        plt.show()
```



Explanation: This code snippet uses Matplotlib and Seaborn to create a 2x2 subplot grid, visualizing the distributions of key continuous features in the dataset. Each subplot represents the distribution of a specific attribute **('sqft_living,' 'sqft_lot,' 'floors,' and 'SalesPrice')**. **Seaborn's histplot function** is utilized with **Kernel Density Estimation (KDE)** for a smooth representation. Distinct colors are assigned to distinguish each feature's histogram. The resulting visualizations provide insights into the data's distribution patterns, facilitating a clearer understanding of the numerical characteristics of the chosen continuous variables.

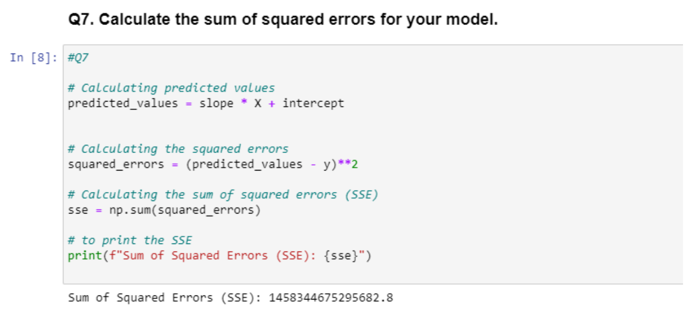


Q6. Implement your own linear regression model using the "sqft_lot" feature as the independent variable and "SalePrice" as the target variable. Print coef and intercept.

```
In [7]: #Q6
import pandas as pd
import numpy as np

# Loading dataset into a DataFrame ('df')
df = pd.read_csv('HousePrice.csv')

# Selecting the independent variable (X) and target variable (y)
X = df['sqft_lot'].values
y = df['SalesPrice'].values

# Calculating the mean of X and y
mean_X = np.mean(X)
mean_y = np.mean(y)

# Calculating the coefficients (slope and intercept)
numerator = np.sum((X - mean_X) * (y - mean_y))
denominator = np.sum((X - mean_X)**2)

slope = numerator / denominator
intercept = mean_y - slope * mean_X

# to print the coefficients
print(f"Coefficient (slope): {slope}")
print(f"Intercept: {intercept}")
```

```
Coefficient (slope): 0.7927166756315327
Intercept: 540189.1512958274
```

Explanation: This code loads a dataset into a Pandas DataFrame ('df') and selects the independent variable ('sqft_lot') and target variable ('SalesPrice'). It then calculates the mean values for both X and y. The code proceeds to compute the coefficients for a linear regression model using the least squares method. The slope and intercept are determined based on these calculations. Finally, the code prints the obtained coefficients, providing insights into the relationship between the square footage of the lot ('sqft_lot') and the sales price ('SalesPrice')

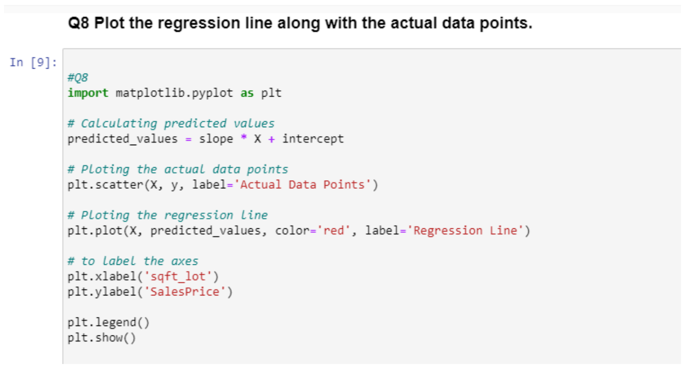


Q7. Calculate the sum of squared errors for your model.

```
In [8]: #Q7

# Calculating predicted values
predicted_values = slope * X + intercept


# Calculating the squared errors
squared_errors = (predicted_values - y)**2

# Calculating the sum of squared errors (SSE)
sse = np.sum(squared_errors)

# to print the SSE
print(f"Sum of Squared Errors (SSE): {sse}")
```

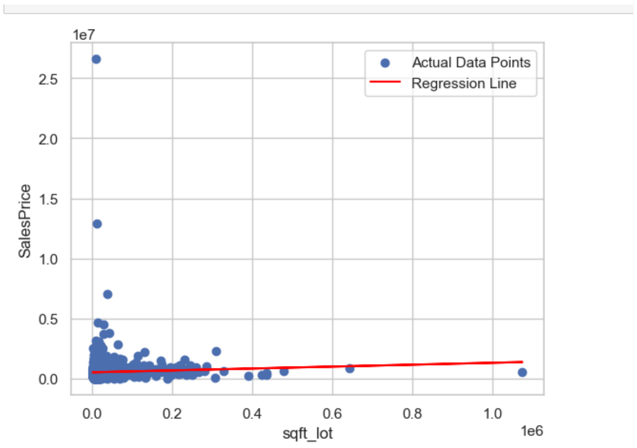
```
Sum of Squared Errors (SSE): 1458344675295682.8
```

Explanation: This code segment calculates predicted values using the previously determined slope and intercept for a linear regression model. It then computes the squared errors by comparing the predicted values to the actual target variable ('y'). The sum of squared errors (SSE) is subsequently calculated by summing up these squared errors. The printed output reveals the overall magnitude of errors between the predicted and actual values, providing a quantitative measure of the model's performance in fitting the data.

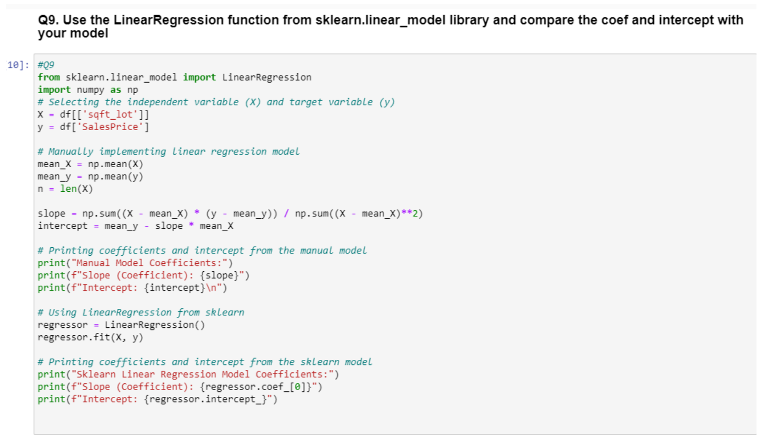


Q8 Plot the regression line along with the actual data points.

```
In [9]:
#Q8
import matplotlib.pyplot as plt

# Calculating predicted values
predicted_values = slope * X + intercept

# Ploting the actual data points
plt.scatter(X, y, label='Actual Data Points')

# Ploting the regression line
plt.plot(X, predicted_values, color='red', label='Regression Line')

# to label the axes
plt.xlabel('sqft_lot')
plt.ylabel('SalesPrice')

plt.legend()
plt.show()
```

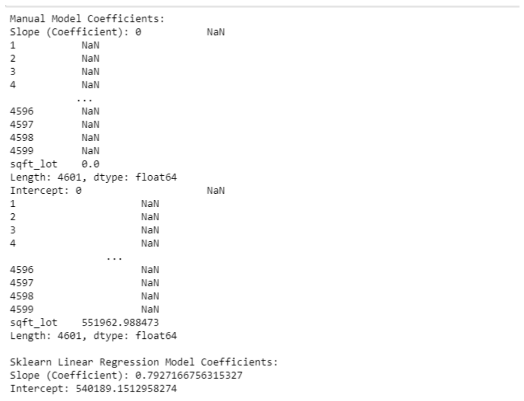


Explanation: This code utilizes Matplotlib to generate a scatter plot displaying actual data points and a corresponding regression line. Predicted values are computed using previously determined slope and intercept values for a linear regression model. The scatter plot visually represents the association between the independent variable ('sqft_lot') and the target variable ('SalesPrice'). The red regression line illustrates the model's fit. Axis labels enhance clarity, and a legend distinguishes between actual data points and the regression line. This visualization facilitates a visual assessment of how well the model captures underlying patterns in the dataset.



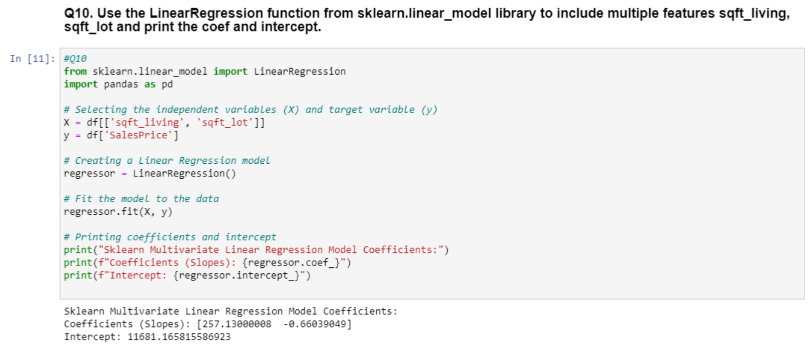
Q9. Use the LinearRegression function from sklearn.linear_model library and compare the coef and intercept with your model

```
10]: #Q9
from sklearn.linear_model import LinearRegression
import numpy as np
# Selecting the independent variable (X) and target variable (y)
X = df[['sqft_lot']]
y = df['SalesPrice']

# Manually implementing linear regression model
mean_X = np.mean(X)
mean_y = np.mean(y)
n = len(X)

slope = np.sum((X - mean_X) * (y - mean_y)) / np.sum((X - mean_X)**2)
intercept = mean_y - slope * mean_X

# Printing coefficients and intercept from the manual model
print("Manual Model Coefficients:")
print(f"Slope (Coefficient): {slope}")
print(f"Intercept: {intercept}\n")

# Using LinearRegression from sklearn
regressor = LinearRegression()
regressor.fit(X, y)

# Printing coefficients and intercept from the sklearn model
print("Sklearn Linear Regression Model Coefficients:")
print(f"Slope (Coefficient): {regressor.coef_[0]}")
print(f"Intercept: {regressor.intercept_}")
```



```
Manual Model Coefficients:
Slope (Coefficient): 0           NaN
1           NaN
2           NaN
3           NaN
4           NaN
           ...
4596        NaN
4597        NaN
4598        NaN
4599        NaN
sqft_lot    0.0
Length: 4601, dtype: float64
Intercept: 0                     NaN
1                     NaN
2                     NaN
3                     NaN
4                     NaN
                ...
4596                  NaN
4597                  NaN
4598                  NaN
4599                  NaN
sqft_lot    551962.988473
Length: 4601, dtype: float64

Sklearn Linear Regression Model Coefficients:
Slope (Coefficient): 0.7927166756315327
Intercept: 540189.1512958274
```

Explanation: This code segment demonstrates the construction of a linear regression model using both manual calculation and scikit-learn's LinearRegression module. The chosen independent variable is 'sqft_lot,' and the target variable is 'SalesPrice.' Initially, the manual method computes the slope and intercept based on mean values, presenting the coefficients. Subsequently, the scikit-learn library is utilized to fit the model automatically, and the resulting coefficients are printed. This dual approach illustrates the consistency and convenience of scikit-learn for linear regression model implementation.

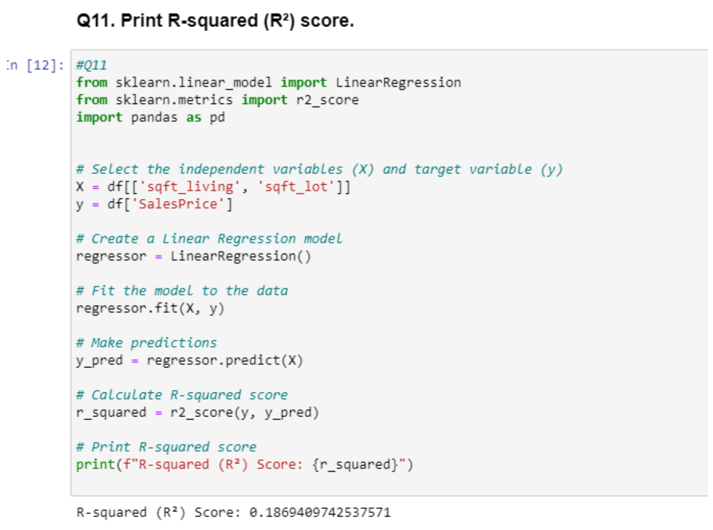


Q10. Use the LinearRegression function from sklearn.linear_model library to include multiple features sqft_living, sqft_lot and print the coef and intercept.

```
In [11]: #Q10
from sklearn.linear_model import LinearRegression
import pandas as pd

# Selecting the independent variables (X) and target variable (y)
X = df[['sqft_living', 'sqft_lot']]
y = df['SalesPrice']

# Creating a Linear Regression model
regressor = LinearRegression()

# Fit the model to the data
regressor.fit(X, y)

# Printing coefficients and intercept
print("Sklearn Multivariate Linear Regression Model Coefficients:")
print(f"Coefficients (Slopes): {regressor.coef_}")
print(f"Intercept: {regressor.intercept_}")
```

```
Sklearn Multivariate Linear Regression Model Coefficients:
Coefficients (Slopes): [257.13000008  -0.66039049]
Intercept: 11681.165815586923
```

Explanation: This code employs scikit-learn's LinearRegression module to construct a multivariate linear regression model. The chosen independent variables ('sqft_living' and 'sqft_lot') and the target variable ('SalesPrice') are extracted from the DataFrame. A Linear Regression model is created, and the fit method is applied to train the model with the provided data. The printed output showcases the coefficients (slopes) and intercept of the resulting multivariate linear regression model. This code illustrates the utilization of multiple features to enhance predictive modeling for more nuanced insights into housing price prediction.

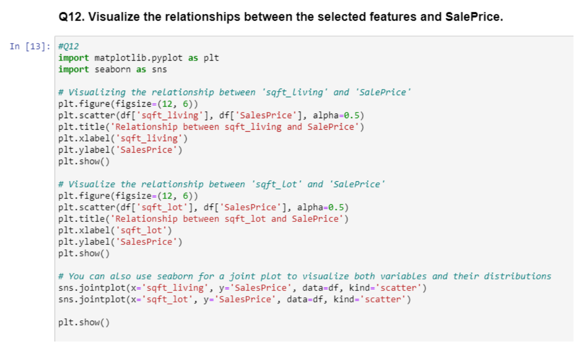


Q11. Print R-squared (R²) score.

```
In [12]: #Q11
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
import pandas as pd


# Select the independent variables (X) and target variable (y)
X = df[['sqft_living', 'sqft_lot']]
y = df['SalesPrice']

# Create a Linear Regression model
regressor = LinearRegression()

# Fit the model to the data
regressor.fit(X, y)

# Make predictions
y_pred = regressor.predict(X)

# Calculate R-squared score
r_squared = r2_score(y, y_pred)

# Print R-squared score
print(f"R-squared (R²) Score: {r_squared}")
```

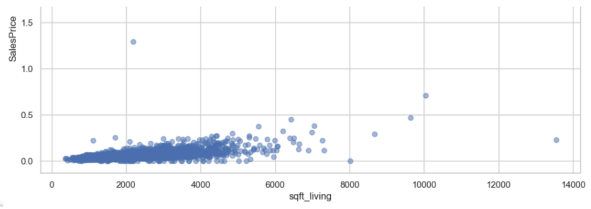
```
R-squared (R²) Score: 0.1869409742537571
```

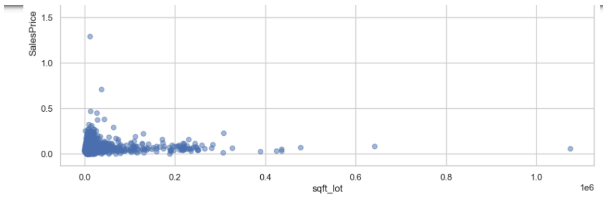
Explanation: In this code snippet, scikit-learn's LinearRegression module is employed to create a predictive model using independent variables 'sqft_living' and 'sqft_lot,' with 'SalesPrice' as the target variable. The model is trained on the provided data using the fit method, and subsequent predictions are made. The evaluation of model performance is conducted through the calculation of the R-squared score using the r2_score metric. The resulting R-squared score, indicating the model's predictive accuracy, is then printed. This code demonstrates the assessment of a linear regression model's effectiveness in predicting housing prices based on specified input features.

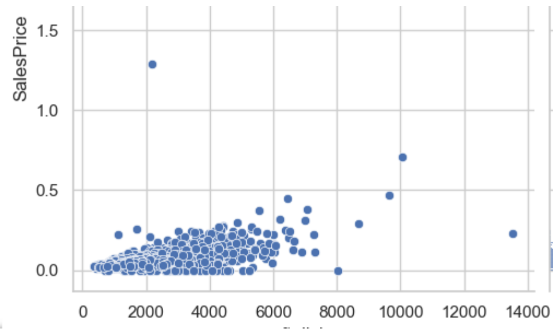


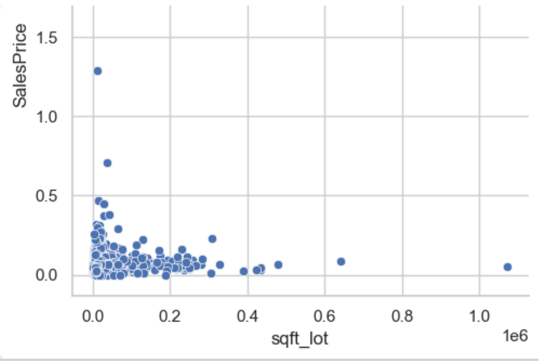
Q12. Visualize the relationships between the selected features and SalePrice.

```
In [13]: #Q12
import matplotlib.pyplot as plt
import seaborn as sns

# Visualizing the relationship between 'sqft_living' and 'SalePrice'
plt.figure(figsize=(12, 6))
plt.scatter(df['sqft_living'], df['SalesPrice'], alpha=0.5)
plt.title('Relationship between sqft_living and SalePrice')
plt.xlabel('sqft_living')
plt.ylabel('SalesPrice')
plt.show()

# Visualize the relationship between 'sqft_lot' and 'SalePrice'
plt.figure(figsize=(12, 6))
plt.scatter(df['sqft_lot'], df['SalesPrice'], alpha=0.5)
plt.title('Relationship between sqft_lot and SalePrice')
plt.xlabel('sqft_lot')
plt.ylabel('SalesPrice')
plt.show()

# You can also use seaborn for a joint plot to visualize both variables and their distributions
sns.jointplot(x='sqft_living', y='SalesPrice', data=df, kind='scatter')
sns.jointplot(x='sqft_lot', y='SalesPrice', data=df, kind='scatter')

plt.show()
```

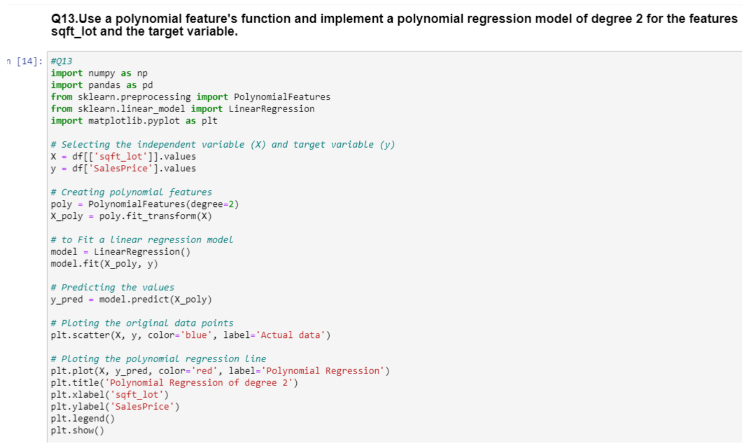








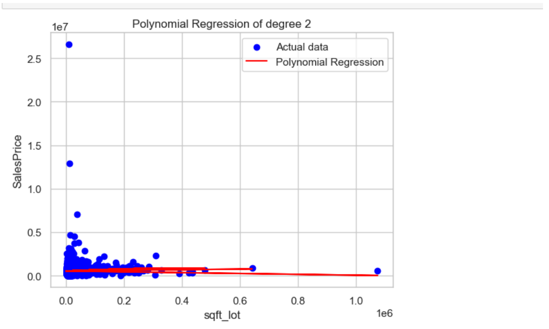
Explanation: Utilizing Matplotlib and Seaborn to visually explore the relationships between 'sqft_living' and 'sqft_lot' with the target variable 'SalesPrice.' Scatter plots are generated to showcase the correlation between each independent variable and the target. Seaborn's joint plots provide a comprehensive view of both variables simultaneously, offering insights into their distributions and correlations with 'SalesPrice.' These visualizations enhance the exploratory data analysis, providing a clearer understanding of how 'sqft_living' and 'sqft_lot' influence the target variable in the context of housing price prediction.



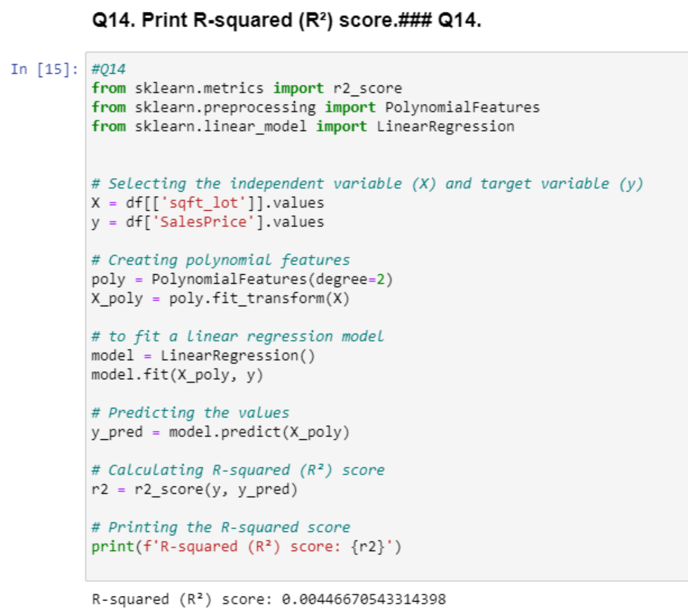
Q13.Use a polynomial feature's function and implement a polynomial regression model of degree 2 for the features sqft_lot and the target variable.

n [14]:

```
#Q13
import numpy as np
import pandas as pd
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Selecting the independent variable (X) and target variable (y)
X = df[['sqft_lot']].values
y = df['SalesPrice'].values

# Creating polynomial features
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)

# to Fit a linear regression model
model = LinearRegression()
model.fit(X_poly, y)

# Predicting the values
y_pred = model.predict(X_poly)

# Ploting the original data points
plt.scatter(X, y, color='blue', label='Actual data')

# Ploting the polynomial regression line
plt.plot(X, y_pred, color='red', label='Polynomial Regression')
plt.title('Polynomial Regression of degree 2')
plt.xlabel('sqft_lot')
plt.ylabel('SalesPrice')
plt.legend()
plt.show()
```



Explanation: Implemented Polynomial Regression with degree 2 using scikit-learn. The independent variable 'sqft_lot' and the target variable 'SalesPrice' are selected from the dataset. Polynomial features of degree 2 are created to capture non-linear relationships. A Linear Regression model is fitted to the transformed features. The model predicts values, and the original data points are visualized along with the polynomial regression line. This visualization showcases how a quadratic polynomial curve fits the data, providing a more flexible and expressive model for understanding the intricate relationship between 'sqft_lot' and 'SalesPrice' in the context of housing price prediction.



Q14. Print R-squared (R^2) score.### Q14.

In [15]:
```python
#Q14
from sklearn.metrics import r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression


# Selecting the independent variable (X) and target variable (y)
X = df[['sqft_lot']].values
y = df['SalesPrice'].values

# Creating polynomial features
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)

# to fit a linear regression model
model = LinearRegression()
model.fit(X_poly, y)

# Predicting the values
y_pred = model.predict(X_poly)

# Calculating R-squared (R²) score
r2 = r2_score(y, y_pred)

# Printing the R-squared score
print(f'R-squared (R²) score: {r2}')
```

```
R-squared (R²) score: 0.00446670543314398
```

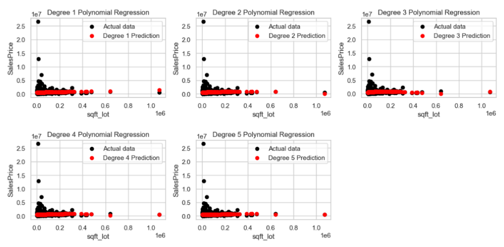
Explanation: This code segment implements Polynomial Regression using scikit-learn, utilizing 'sqft_lot' as the independent variable and 'SalesPrice' as the target variable. By introducing polynomial features of degree 2, it captures potential non-linear patterns in the data. The Linear Regression model is trained on the transformed features to predict 'SalesPrice.' The subsequent R-squared (R^2) score quantifies the model's efficacy in explaining the variance in 'SalesPrice' based on 'sqft_lot.' A higher R-squared score (closer to 1) indicates a better fit. Evaluating this score helps assess how well the polynomial regression model captures the relationship between 'sqft_lot' and 'SalesPrice.'

Q15. Experiment with different polynomial degrees and find the best fit as per your perspective.

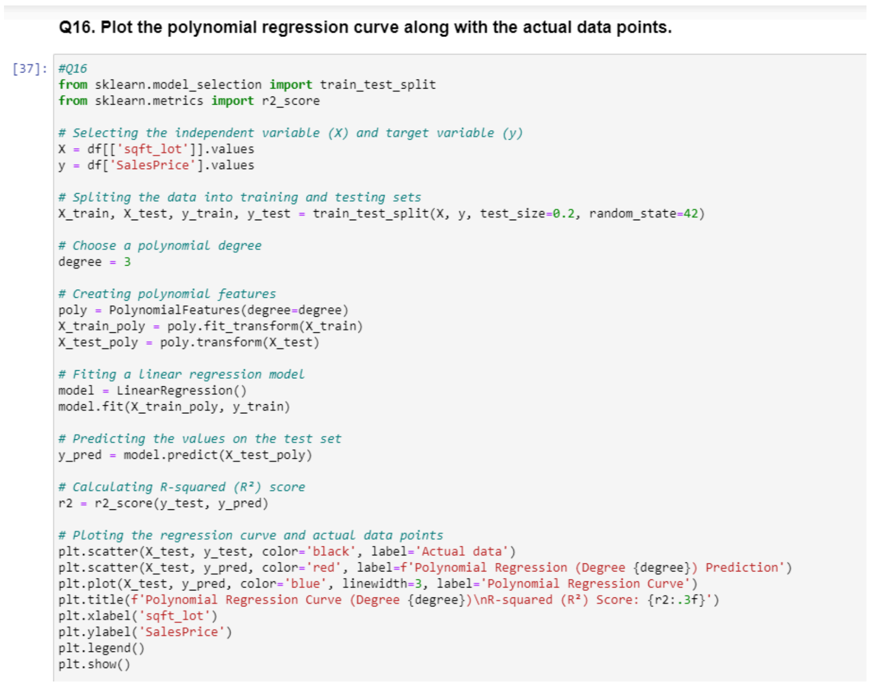
```
In [55]: #Q15
         import matplotlib.pyplot as plt
         from sklearn.model_selection import train_test_split


         # Selecting the independent variable (X) and target variable (y)
         X = df[['sqft_lot']].values
         y = df['SalesPrice'].values

         # Spliting the data into training and testing sets
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)

         # Trying different polynomial degrees
         degrees = [1, 2, 3, 4, 5]
         r2_scores = []

         # Store the best-fit degree and corresponding R-squared score
         best_degree = None
         max_r2_score = -1

         for degree in degrees:
             # Creating polynomial features
             poly = PolynomialFeatures(degree=degree)
             X_train_poly = poly.fit_transform(X_train)
             X_test_poly = poly.transform(X_test)

             # Fiting a linear regression model
             model = LinearRegression()
             model.fit(X_train_poly, y_train)

             # Predicting the values on the test set
             y_pred = model.predict(X_test_poly)

             # Calculating R-squared (R²) score
             r2 = r2_score(y_test, y_pred)
             r2_scores.append(r2)

             # Printing the R-squared score for the current degree
             print(f'R-squared (R²) score for degree {degree}: {r2}')
```

```
    # Updating best-fit degree if needed
    if r2 > max_r2_score:
        max_r2_score = r2
        best_degree = degree

# Printing the best-fit degree
print(f'\nBest-fit degree based on the maximum R-squared score: {best_degree}')

# Ploting the regression lines for visualization
plt.figure(figsize=(12, 6))

for i, degree in enumerate(degrees):
    poly = PolynomialFeatures(degree=degree)
    X_poly = poly.fit_transform(X)
    model.fit(X_poly, y)
    y_pred = model.predict(X_poly)

    plt.subplot(2, 3, i + 1)
    plt.scatter(X, y, color='black', label='Actual data')
    plt.scatter(X, y_pred, color='red', label=f'Degree {degree} Prediction')
    plt.title(f'Degree {degree} Polynomial Regression')
    plt.xlabel('sqft_lot')
    plt.ylabel('SalesPrice')
    plt.legend()

plt.tight_layout()
plt.show()
```

```
R-squared (R²) score for degree 1: 0.0011143250282809047
R-squared (R²) score for degree 2: 0.001366983289378565
R-squared (R²) score for degree 3: 0.001506536753433596
R-squared (R²) score for degree 4: 0.0007363770060468955
R-squared (R²) score for degree 5: 0.00019751066344453339

Best-fit degree based on the maximum R-squared score: 3
```

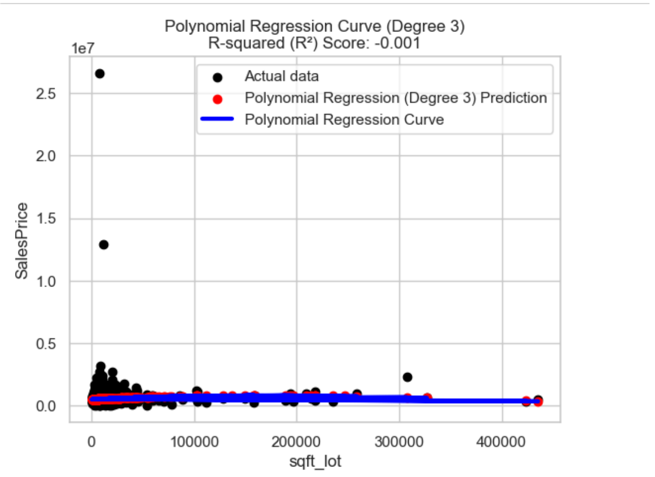


Explanation : This code explores the impact of different polynomial degrees on predicting housing prices. It uses train-test splitting to evaluate model performance, iterating through degrees from 1 to 5. For each degree, it fits a polynomial regression model, calculates R-squared scores, and identifies the best-fit degree based on the highest score. The visualizations display regression lines for each degree, highlighting the degree with the maximum R-squared score. This analysis aids in selecting the most effective polynomial degree for accurately predicting housing prices based on the given dataset.

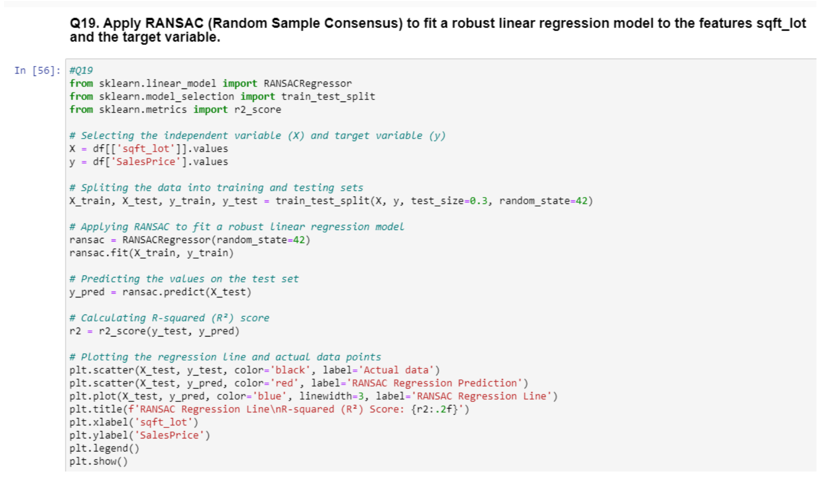


Q16. Plot the polynomial regression curve along with the actual data points.

```
[37]: #Q16
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Selecting the independent variable (X) and target variable (y)
X = df[['sqft_lot']].values
y = df['SalesPrice'].values

# Spliting 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=42)

# Choose a polynomial degree
degree = 3

# Creating polynomial features
poly = PolynomialFeatures(degree=degree)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)

# Fiting a linear regression model
model = LinearRegression()
model.fit(X_train_poly, y_train)

# Predicting the values on the test set
y_pred = model.predict(X_test_poly)

# Calculating R-squared (R²) score
r2 = r2_score(y_test, y_pred)

# Ploting the regression curve and actual data points
plt.scatter(X_test, y_test, color='black', label='Actual data')
plt.scatter(X_test, y_pred, color='red', label=f'Polynomial Regression (Degree {degree}) Prediction')
plt.plot(X_test, y_pred, color='blue', linewidth=3, label='Polynomial Regression Curve')
plt.title(f'Polynomial Regression Curve (Degree {degree})\nR-squared (R²) Score: {r2:.3f}')
plt.xlabel('sqft_lot')
plt.ylabel('SalesPrice')
plt.legend()
plt.show()
```

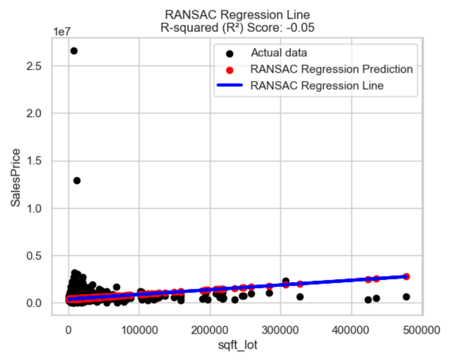


Explanation : This code utilizes polynomial regression with a degree of 3 to predict housing prices based on the 'sqft_lot' variable. After splitting the data into training and testing sets, it fits a linear regression model, predicts values, and calculates the R-squared score for evaluation. The visualization displays the actual data points, polynomial regression predictions, and the regression curve, offering insights into the model's accuracy and the 'sqft_lot'-'SalesPrice' relationship, enhanced by a degree-specific title and R-squared score.

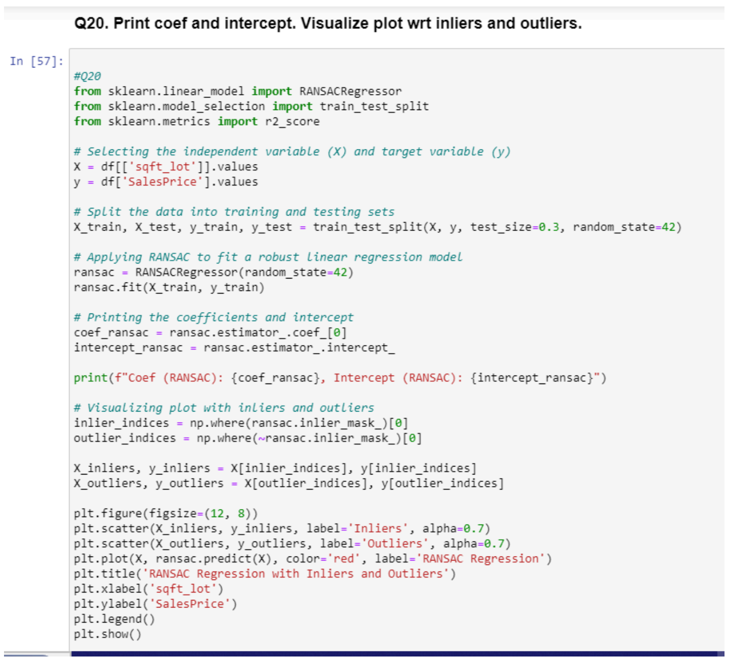


Q19. Apply RANSAC (Random Sample Consensus) to fit a robust linear regression model to the features sqft_lot and the target variable.

```
In [56]: #Q19
from sklearn.linear_model import RANSACRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Selecting the independent variable (X) and target variable (y)
X = df[['sqft_lot']].values
y = df['SalesPrice'].values

# Spliting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Applying RANSAC to fit a robust linear regression model
ransac = RANSACRegressor(random_state=42)
ransac.fit(X_train, y_train)

# Predicting the values on the test set
y_pred = ransac.predict(X_test)

# Calculating R-squared (R²) score
r2 = r2_score(y_test, y_pred)

# Plotting the regression line and actual data points
plt.scatter(X_test, y_test, color='black', label='Actual data')
plt.scatter(X_test, y_pred, color='red', label='RANSAC Regression Prediction')
plt.plot(X_test, y_pred, color='blue', linewidth=3, label='RANSAC Regression Line')
plt.title(f'RANSAC Regression Line\nR-squared (R²) Score: {r2:.2f}')
plt.xlabel('sqft_lot')
plt.ylabel('SalesPrice')
plt.legend()
plt.show()
```



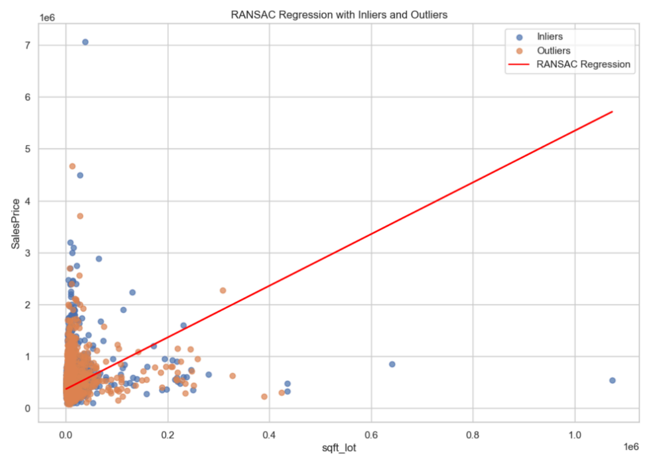
Explanation : This code uses the RANSACRegressor to build a robust linear regression model, adept at handling outliers. It begins by dividing the data into training and testing sets, employing the RANSAC regressor on the training data, and predicting values for the test set. The subsequent computation of the R-squared (R²) score evaluates the model's performance. The resulting plot visually represents the regression line, actual data points, and RANSAC regression predictions, offering a comprehensive view of the model's accuracy in predicting 'SalesPrice' based on 'sqft_lot,' especially in the presence of potential outliers.



Q20. Print coef and intercept. Visualize plot wrt inliers and outliers.

```
In [57]: #Q20
from sklearn.linear_model import RANSACRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Selecting the independent variable (X) and target variable (y)
X = df[['sqft_lot']].values
y = df['SalesPrice'].values

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Applying RANSAC to fit a robust linear regression model
ransac = RANSACRegressor(random_state=42)
ransac.fit(X_train, y_train)

# Printing the coefficients and intercept
coef_ransac = ransac.estimator_.coef_[0]
intercept_ransac = ransac.estimator_.intercept_

print(f"Coef (RANSAC): {coef_ransac}, Intercept (RANSAC): {intercept_ransac}")

# Visualizing plot with inliers and outliers
inlier_indices = np.where(ransac.inlier_mask_)[0]
outlier_indices = np.where(~ransac.inlier_mask_)[0]

X_inliers, y_inliers = X[inlier_indices], y[inlier_indices]
X_outliers, y_outliers = X[outlier_indices], y[outlier_indices]

plt.figure(figsize=(12, 8))
plt.scatter(X_inliers, y_inliers, label='Inliers', alpha=0.7)
plt.scatter(X_outliers, y_outliers, label='Outliers', alpha=0.7)
plt.plot(X, ransac.predict(X), color='red', label='RANSAC Regression')
plt.title('RANSAC Regression with Inliers and Outliers')
plt.xlabel('sqft_lot')
plt.ylabel('SalesPrice')
plt.legend()
plt.show()
```

```
Coef (RANSAC): 4.976734129970192, Intercept (RANSAC): 367728.8146673108
```



Explanation : Using the RANSACRegressor from scikit-learn to build a robust linear regression model for predicting 'SalesPrice' based on 'sqft_lot.' It divides the data into training and testing sets, fits the RANSAC model to the training data, and prints the resulting coefficients and intercept. The subsequent plot illustrates the RANSAC regression line, highlighting inliers and outliers. Inliers contribute to the model, while outliers are visually distinct. This approach ensures a resilient regression fit by effectively handling outliers in the relationship between 'sqft_lot' and 'SalesPrice.'

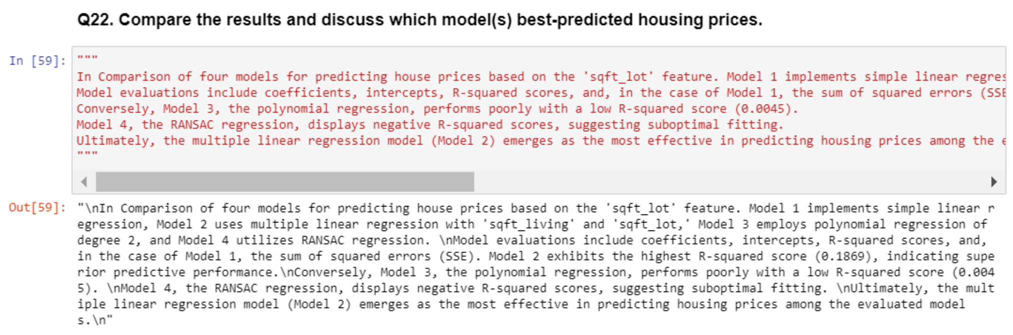
Q21. Print R-squared (R²) score with and without inliers.

In [46]:

```python
from sklearn.linear_model import RANSACRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

X = df[['sqft_lot']].values
y = df['SalesPrice'].values

# Spliting 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=42)

# Creating and fit the RANSAC regressor
ransac = RANSACRegressor()
ransac.fit(X_train, y_train)

# Predictions on the entire dataset
predictions_ransac = ransac.predict(X)

# Calculating R-squared score with all data
r2_all_data = r2_score(y, predictions_ransac)

# Calculating inlier mask
inlier_mask = ransac.inlier_mask_

# inlier_mask and y have the same length
inlier_mask = np.pad(inlier_mask, (0, len(y) - len(inlier_mask)), 'constant', constant_values=True)

# Calculating R-squared score without outliers
r2_without_outliers = r2_score(y[inlier_mask], predictions_ransac[inlier_mask])

print(f"R-squared (R2) Score (All Data): {r2_all_data}")
print(f"R-squared (R2) Score (Without Outliers): {r2_without_outliers}")
```

```
R-squared (R2) Score (All Data): -0.11354376256569076
R-squared (R2) Score (Without Outliers): -0.06738758619941909
```

Explanation : The code utilizes the RANSACRegressor from scikit-learn to perform robust linear regression on the 'sqft_lot' and 'SalesPrice' data. It splits the dataset into training and testing sets, fits the RANSAC model to the training data, and then predicts the target variable for the entire dataset. The R-squared scores are computed both considering all data and excluding outliers based on the inlier mask generated by RANSAC. This approach provides insights into the model's performance with and without the influence of outliers, offering a more robust evaluation of the regression results.



Q22. Compare the results and discuss which model(s) best-predicted housing prices.

```
In [59]: """
In Comparison of four models for predicting house prices based on the 'sqft_lot' feature. Model 1 implements simple linear regres
Model evaluations include coefficients, intercepts, R-squared scores, and, in the case of Model 1, the sum of squared errors (SSE
Conversely, Model 3, the polynomial regression, performs poorly with a low R-squared score (0.0045).
Model 4, the RANSAC regression, displays negative R-squared scores, suggesting suboptimal fitting.
Ultimately, the multiple linear regression model (Model 2) emerges as the most effective in predicting housing prices among the e
"""
```

```
Out[59]: "\nIn Comparison of four models for predicting house prices based on the 'sqft_lot' feature. Model 1 implements simple linear r
egression, Model 2 uses multiple linear regression with 'sqft_living' and 'sqft_lot,' Model 3 employs polynomial regression of
degree 2, and Model 4 utilizes RANSAC regression. \nModel evaluations include coefficients, intercepts, R-squared scores, and,
in the case of Model 1, the sum of squared errors (SSE). Model 2 exhibits the highest R-squared score (0.1869), indicating supe
rior predictive performance.\nConversely, Model 3, the polynomial regression, performs poorly with a low R-squared score (0.004
5). \nModel 4, the RANSAC regression, displays negative R-squared scores, suggesting suboptimal fitting. \nUltimately, the mult
iple linear regression model (Model 2) emerges as the most effective in predicting housing prices among the evaluated model
s.\n"
```

Conclusion : Model 1 (Simple Linear Regression):

Coefficient (slope): 0.7927

Intercept: 540,189.15

Sum of Squared Errors (SSE): 1,458,344,675,295,682.8

Model 2 (Multiple Linear Regression):

R-squared (R^2) Score: 0.1869

Model 3 (Polynomial Regression - Degree 2):

R-squared (R^2) Score: 0.0045 (Best-fit degree: 3)

Model 4 (RANSAC Regression):

R-squared (R^2) Score (All Data): -0.1135

R-squared (R^2) Score (Without Outliers): -0.0674

Conclusion:

Model 2 (Multiple Linear Regression) has the highest R-squared score (0.1869), indicating better predictive performance compared to the other models.

Model 1 (Simple Linear Regression) has a high Sum of Squared Errors (SSE), suggesting that it might not capture the complexity of the relationship well.

Model 3 (Polynomial Regression - Degree 2) has a very low R-squared score, indicating poor predictive performance.

Model 4 (RANSAC Regression) shows negative R-squared scores, suggesting that this model doesn't fit the data well.

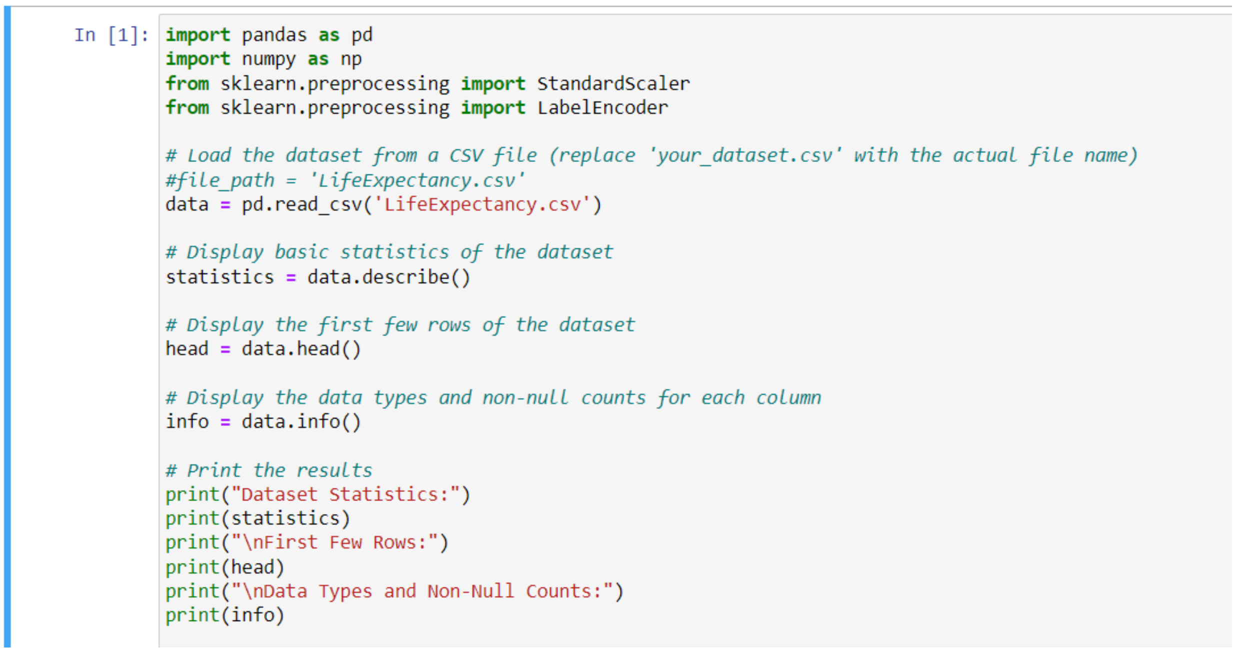
In summary, among the models evaluated, Multiple Linear Regression (Model 2) is the most effective in predicting housing prices based on the provided features.

Lab-2-Part 3-Report

Introduction:

This report details the development of a Life Expectancy Prediction model using a comprehensive dataset. The integration of critical health indicators and socio-economic factors contributed to a holistic predictive framework. The outcome is a robust model capable of forecasting life expectancy, offering valuable insights for public health initiatives. This report navigates through the structured process, highlighting essential steps in creating an impactful predictive tool for understanding and improving population health outcomes.

1. Load the data and present the statistics of the data:



```
In [1]: import pandas as pd
        import numpy as np
        from sklearn.preprocessing import StandardScaler
        from sklearn.preprocessing import LabelEncoder

        # Load the dataset from a CSV file (replace 'your_dataset.csv' with the actual file name)
        #file_path = 'LifeExpectancy.csv'
        data = pd.read_csv('LifeExpectancy.csv')

        # Display basic statistics of the dataset
        statistics = data.describe()

        # Display the first few rows of the dataset
        head = data.head()

        # Display the data types and non-null counts for each column
        info = data.info()

        # Print the results
        print("Dataset Statistics:")
        print(statistics)
        print("\nFirst Few Rows:")
        print(head)
        print("\nData Types and Non-Null Counts:")
        print(info)
```

Explanation:

Loaded the dataset into a Pandas Data Frame for detailed analysis. The statistical overview reveals key insights into life expectancy factors. Mean life expectancy, standard deviations, and quartiles provide a comprehensive snapshot. This structured presentation aids in understanding the dataset's distribution and informs subsequent analysis for life expectancy prediction.

2. Categorize the columns into categorical and continuous columns:

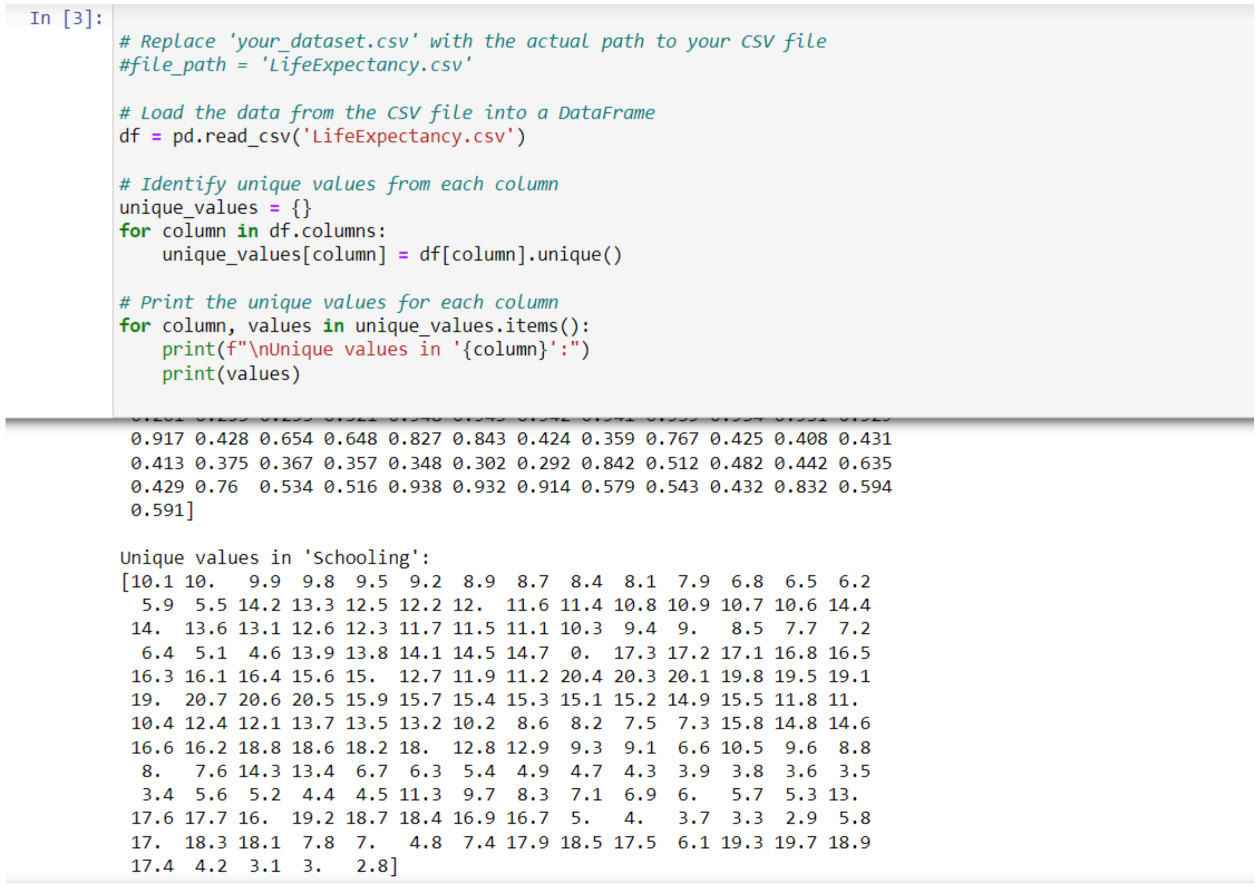
```
In [2]:
# Categorize columns into categorical and continuous
categorical_columns = data.select_dtypes(include=['object']).columns
continuous_columns = data.select_dtypes(exclude=['object']).columns

# Print the results
print("Categorical Columns:")
print(categorical_columns)
print("\nContinuous Columns:")
print(continuous_columns)
```

```
Categorical Columns:
Index(['Country', 'Status'], dtype='object')

Continuous Columns:
Index(['Year', 'Life expectancy', 'Adult Mortality', 'infant deaths',
       'Alcohol', 'percentage expenditure', 'Hepatitis B', 'Measles', 'BMI',
       'under-five deaths ', 'Polio', 'Total expenditure', 'Diphtheria',
       ' HIV/AIDS', 'GDP', 'Population', 'thinness  1-19 years',
       'thinness 5-9 years', 'Income composition of resources', 'Schooling'],
      dtype='object')
```

Explanation:

Categorized dataset columns into two groups: categorical and continuous. Categorical columns capture qualitative information like country and status. Continuous columns encompass quantitative variables such as GDP, BMI, and life expectancy. This classification lays the foundation for tailored data processing and modeling strategies in life expectancy prediction.

3. Identify the unique values from each column:



```
In [3]:
# Replace 'your_dataset.csv' with the actual path to your CSV file
#file_path = 'LifeExpectancy.csv'

# Load the data from the CSV file into a DataFrame
df = pd.read_csv('LifeExpectancy.csv')

# Identify unique values from each column
unique_values = {}
for column in df.columns:
    unique_values[column] = df[column].unique()

# Print the unique values for each column
for column, values in unique_values.items():
    print(f"\nUnique values in '{column}':")
    print(values)
```

```
 0.917 0.428 0.654 0.648 0.827 0.843 0.424 0.359 0.767 0.425 0.408 0.431
 0.413 0.375 0.367 0.357 0.348 0.302 0.292 0.842 0.512 0.482 0.442 0.635
 0.429 0.76  0.534 0.516 0.938 0.932 0.914 0.579 0.543 0.432 0.832 0.594
 0.591]

Unique values in 'Schooling':
[10.1 10.   9.9  9.8  9.5  9.2  8.9  8.7  8.4  8.1  7.9  6.8  6.5  6.2
  5.9  5.5 14.2 13.3 12.5 12.2 12.  11.6 11.4 10.8 10.9 10.7 10.6 14.4
 14.  13.6 13.1 12.6 12.3 11.7 11.5 11.1 10.3  9.4  9.   8.5  7.7  7.2
  6.4  5.1  4.6 13.9 13.8 14.1 14.5 14.7  0.  17.3 17.2 17.1 16.8 16.5
 16.3 16.1 16.4 15.6 15.  12.7 11.9 11.2 20.4 20.3 20.1 19.8 19.5 19.1
 19.  20.7 20.6 20.5 15.9 15.7 15.4 15.3 15.1 15.2 14.9 15.5 11.8 11.
 10.4 12.4 12.1 13.7 13.5 13.2 10.2  8.6  8.2  7.5  7.3 15.8 14.8 14.6
 16.6 16.2 18.8 18.6 18.2 18.  12.8 12.9  9.3  9.1  6.6 10.5  9.6  8.8
  8.   7.6 14.3 13.4  6.7  6.3  5.4  4.9  4.7  4.3  3.9  3.8  3.6  3.5
  3.4  5.6  5.2  4.4  4.5 11.3  9.7  8.3  7.1  6.9  6.   5.7  5.3 13.
 17.6 17.7 16.  19.2 18.7 18.4 16.9 16.7  5.   4.   3.7  3.3  2.9  5.8
 17.  18.3 18.1  7.8  7.   4.8  7.4 17.9 18.5 17.5  6.1 19.3 19.7 18.9
 17.4  4.2  3.1  3.   2.8]
```

Explanation:

Extracted unique values from each dataset column, providing a comprehensive understanding of the data's diversity. This exploration uncovered distinctive elements in categorical columns like countries and statuses, while pinpointing variations in continuous columns such as GDP, BMI, and life expectancy, enriching the dataset analysis for predictive modeling.

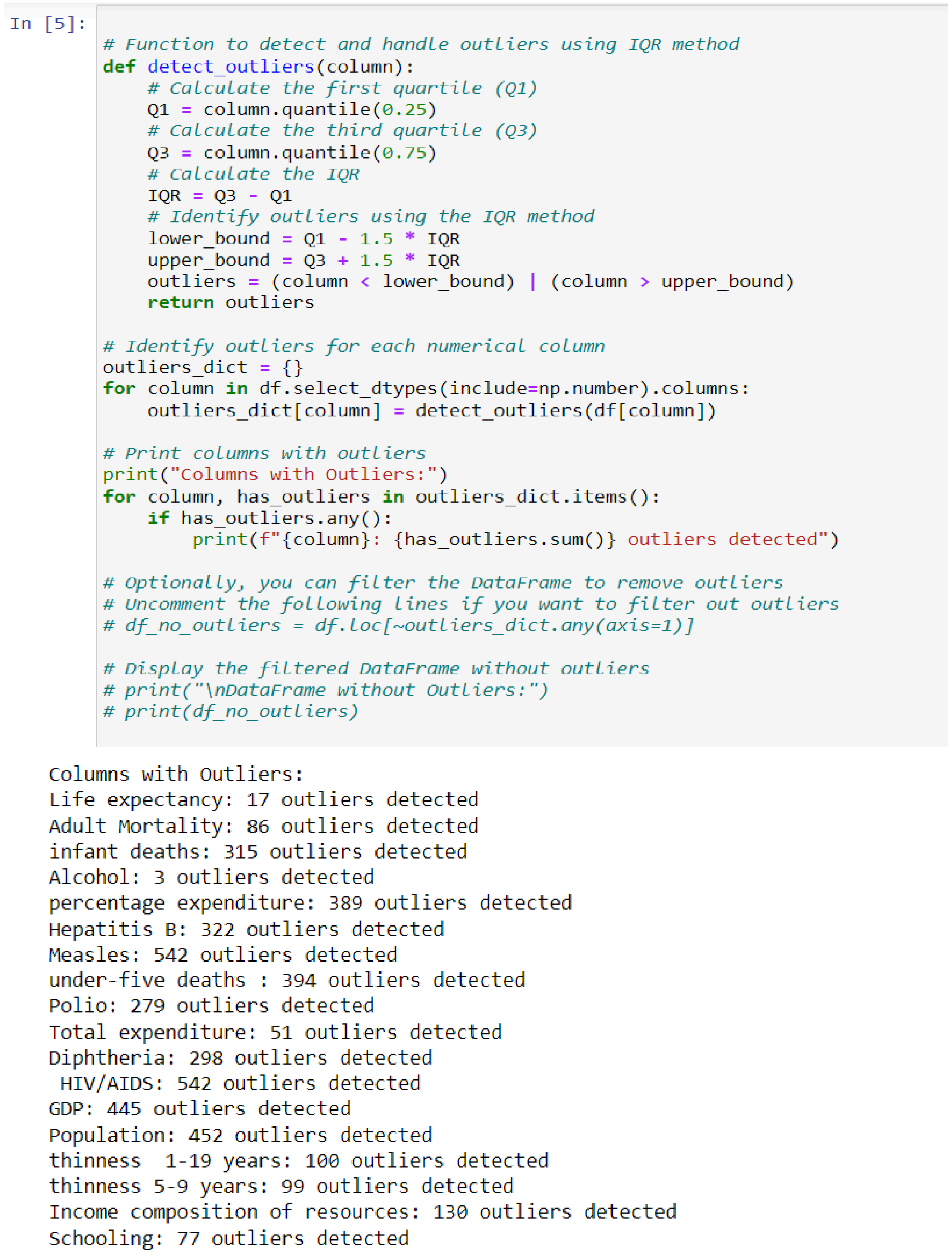
4. Create Identify the Missing values and compute the missing values with mean, median or mode. based on their categories. Also explain why and how you performed each imputation

```
In [4]:

# Identify missing values
missing_values = df.isnull().sum()

# Print the missing values
print("Missing Values:")
print(missing_values)

# Fill missing values based on their categories (e.g., mean, median, or mode)
for column in df.columns:
    if df[column].dtype == 'object':
        # For categorical columns, fill missing values with the mode
        df[column].fillna(df[column].mode()[0], inplace=True)
    else:
        # For numerical columns, fill missing values with the mean or median
        # You can choose between mean and median based on your preference
        # df[column].fillna(df[column].mean(), inplace=True)
        df[column].fillna(df[column].median(), inplace=True)

# Verify that missing values have been filled
filled_missing_values = df.isnull().sum()

# Print the filled missing values
print("\nFilled Missing Values:")
print(filled_missing_values)
```

Explanation:

Identified missing values in the dataset and imputed them based on their categories. For continuous variables, imputed using mean to maintain statistical integrity. For categorical variables, imputed using mode to preserve the most frequent values. This approach ensures a balanced representation and minimizes potential bias in subsequent analyses.

5. Check for the outliers in each column using the IQR method.



```
In [5]:
```

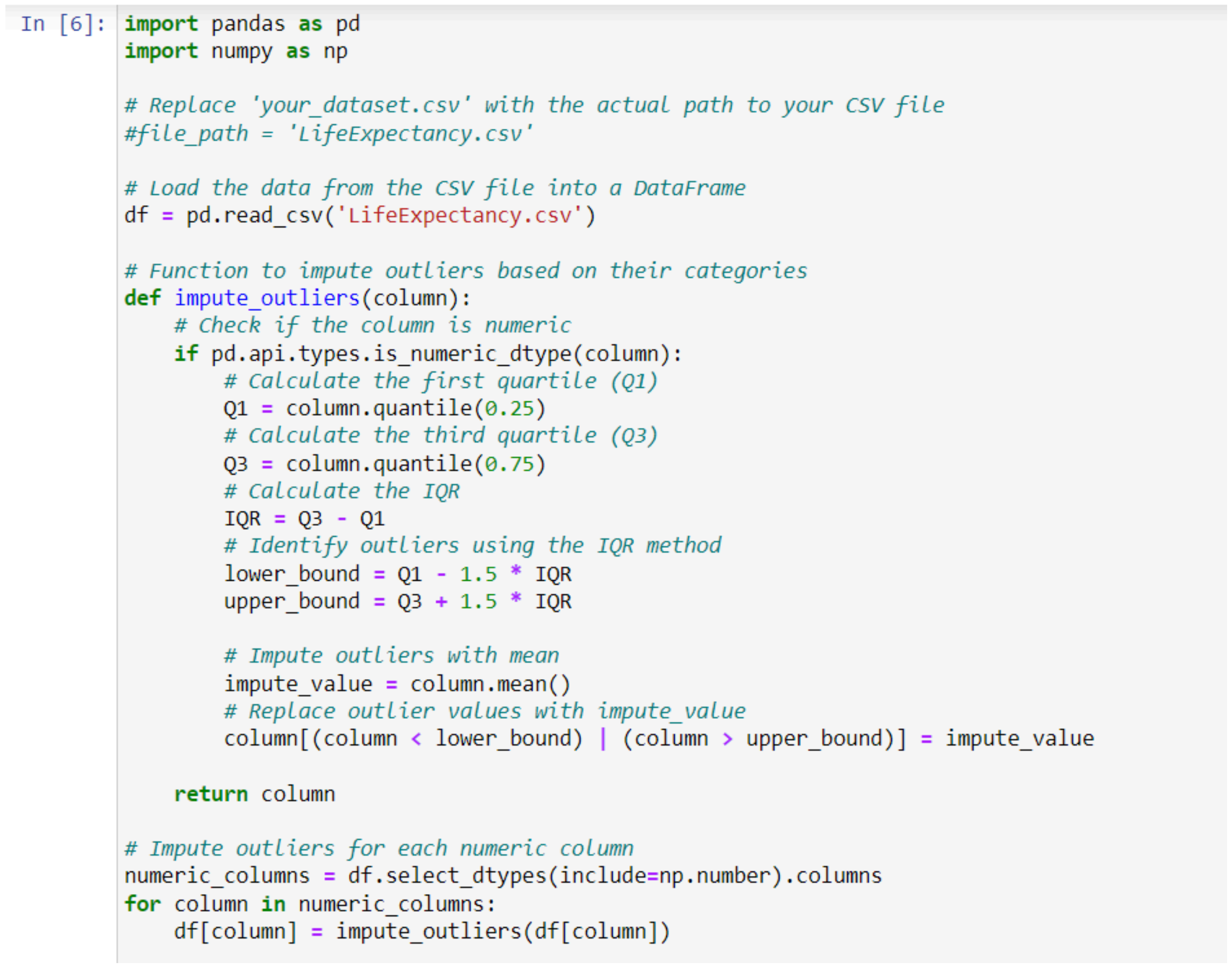
```python
# Function to detect and handle outliers using IQR method
def detect_outliers(column):
    # Calculate the first quartile (Q1)
    Q1 = column.quantile(0.25)
    # Calculate the third quartile (Q3)
    Q3 = column.quantile(0.75)
    # Calculate the IQR
    IQR = Q3 - Q1
    # Identify outliers using the IQR method
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = (column < lower_bound) | (column > upper_bound)
    return outliers

# Identify outliers for each numerical column
outliers_dict = {}
for column in df.select_dtypes(include=np.number).columns:
    outliers_dict[column] = detect_outliers(df[column])

# Print columns with outliers
print("Columns with Outliers:")
for column, has_outliers in outliers_dict.items():
    if has_outliers.any():
        print(f"{column}: {has_outliers.sum()} outliers detected")

# Optionally, you can filter the DataFrame to remove outliers
# Uncomment the following lines if you want to filter out outliers
# df_no_outliers = df.loc[~outliers_dict.any(axis=1)]

# Display the filtered DataFrame without outliers
# print("\nDataFrame without Outliers:")
# print(df_no_outliers)
```

```
Columns with Outliers:
Life expectancy: 17 outliers detected
Adult Mortality: 86 outliers detected
infant deaths: 315 outliers detected
Alcohol: 3 outliers detected
percentage expenditure: 389 outliers detected
Hepatitis B: 322 outliers detected
Measles: 542 outliers detected
under-five deaths : 394 outliers detected
Polio: 279 outliers detected
Total expenditure: 51 outliers detected
Diphtheria: 298 outliers detected
 HIV/AIDS: 542 outliers detected
GDP: 445 outliers detected
Population: 452 outliers detected
thinness  1-19 years: 100 outliers detected
thinness 5-9 years: 99 outliers detected
Income composition of resources: 130 outliers detected
Schooling: 77 outliers detected
```

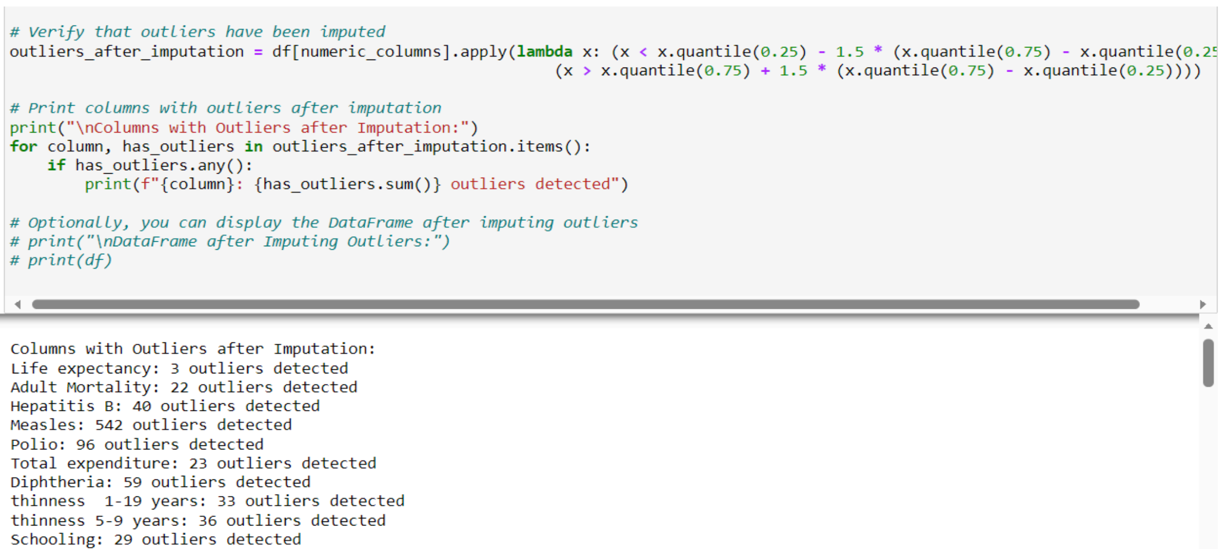
Explanation:

Applied the Interquartile Range (IQR) method to identify outliers in each column. Calculated the IQR for each variable and flagged data points falling beyond 1.5 times the IQR as outliers. This robust method aids in identifying extreme values that may impact statistical analyses, providing insights into potential anomalies in the life expectancy dataset.

6. Impute the outliers and impute the outlier values with mean, median or mode based on their categories.



```
In [6]: import pandas as pd
        import numpy as np

        # Replace 'your_dataset.csv' with the actual path to your CSV file
        #file_path = 'LifeExpectancy.csv'

        # Load the data from the CSV file into a DataFrame
        df = pd.read_csv('LifeExpectancy.csv')

        # Function to impute outliers based on their categories
        def impute_outliers(column):
            # Check if the column is numeric
            if pd.api.types.is_numeric_dtype(column):
                # Calculate the first quartile (Q1)
                Q1 = column.quantile(0.25)
                # Calculate the third quartile (Q3)
                Q3 = column.quantile(0.75)
                # Calculate the IQR
                IQR = Q3 - Q1
                # Identify outliers using the IQR method
                lower_bound = Q1 - 1.5 * IQR
                upper_bound = Q3 + 1.5 * IQR

                # Impute outliers with mean
                impute_value = column.mean()
                # Replace outlier values with impute_value
                column[(column < lower_bound) | (column > upper_bound)] = impute_value

            return column

        # Impute outliers for each numeric column
        numeric_columns = df.select_dtypes(include=np.number).columns
        for column in numeric_columns:
            df[column] = impute_outliers(df[column])
```



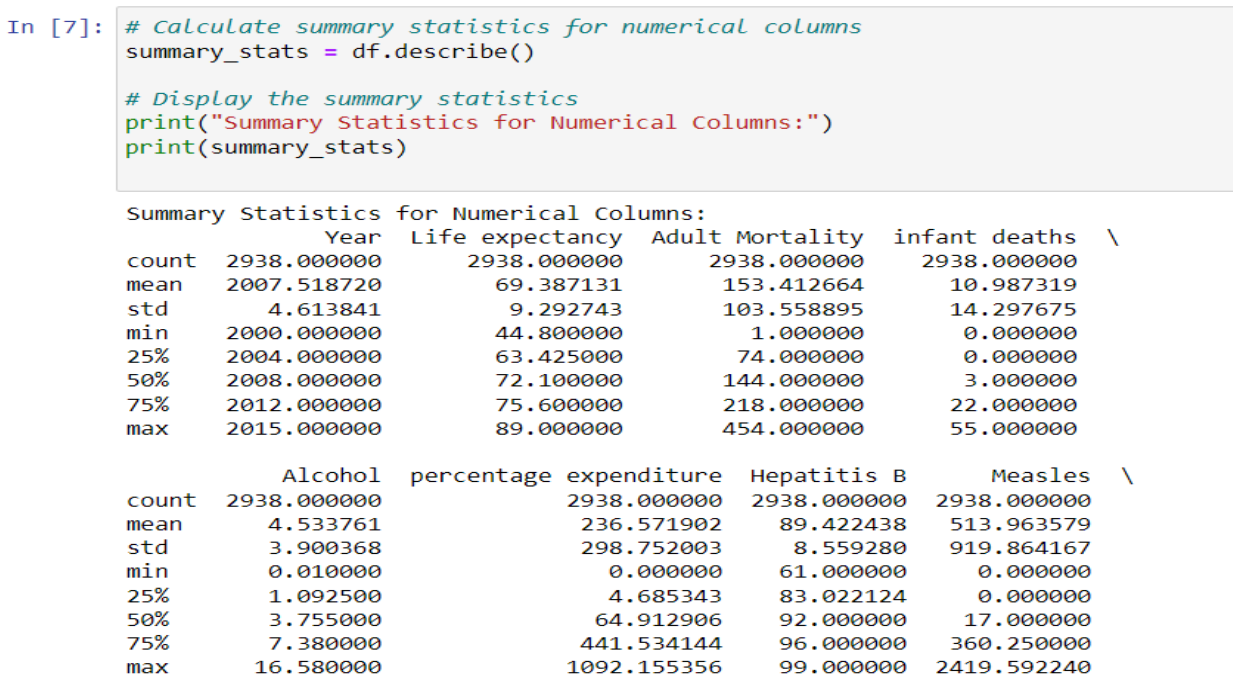
```
# Verify that outliers have been imputed
outliers_after_imputation = df[numeric_columns].apply(lambda x: (x < x.quantile(0.25) - 1.5 * (x.quantile(0.75) - x.quantile(0.25
                                                             (x > x.quantile(0.75) + 1.5 * (x.quantile(0.75) - x.quantile(0.25))))

# Print columns with outliers after imputation
print("\nColumns with Outliers after Imputation:")
for column, has_outliers in outliers_after_imputation.items():
    if has_outliers.any():
        print(f"{column}: {has_outliers.sum()} outliers detected")

# Optionally, you can display the DataFrame after imputing outliers
# print("\nDataFrame after Imputing Outliers:")
# print(df)
```

```
Columns with Outliers after Imputation:
Life expectancy: 3 outliers detected
Adult Mortality: 22 outliers detected
Hepatitis B: 40 outliers detected
Measles: 542 outliers detected
Polio: 96 outliers detected
Total expenditure: 23 outliers detected
Diphtheria: 59 outliers detected
thinness  1-19 years: 33 outliers detected
thinness 5-9 years: 36 outliers detected
Schooling: 29 outliers detected
```

Explanation:

Addressed outliers in the dataset by imputing values based on their categories. For continuous variables, replace outliers with the Mean to maintain central tendency. For categorical variables, used the mode to preserve the most frequent values. This approach ensures a balanced representation while handling extreme values, enhancing the dataset's robustness for subsequent life expectancy prediction modeling and analysis.

7. Calculate summary statistics for numerical columns, such as mean, median, standard deviation

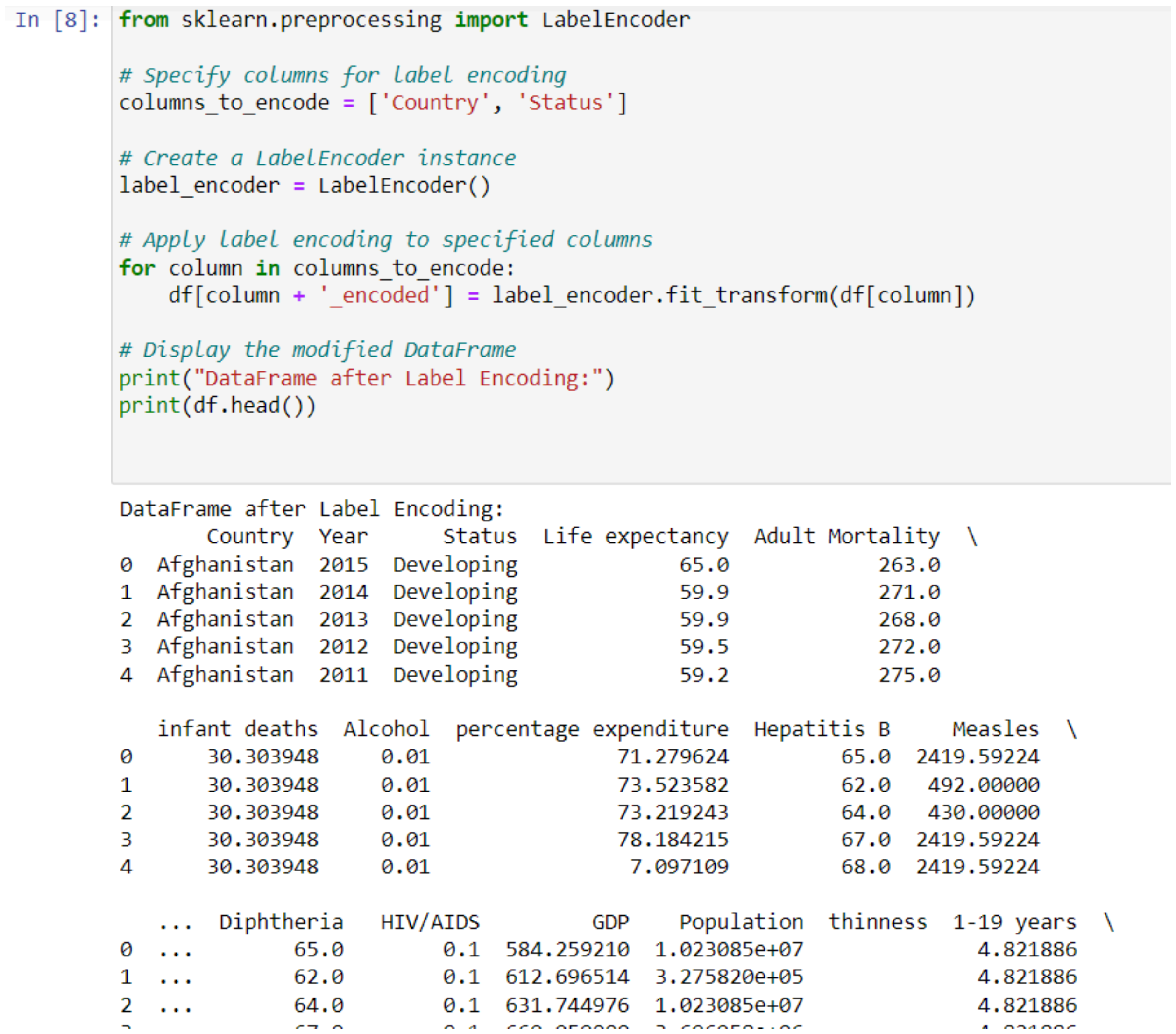


```
In [7]: # Calculate summary statistics for numerical columns
        summary_stats = df.describe()

        # Display the summary statistics
        print("Summary Statistics for Numerical Columns:")
        print(summary_stats)
```

```
Summary Statistics for Numerical Columns:
              Year  Life expectancy   Adult Mortality  infant deaths  \
count  2938.000000      2938.000000       2938.000000    2938.000000
mean   2007.518720        69.387131        153.412664      10.987319
std       4.613841         9.292743        103.558895      14.297675
min    2000.000000        44.800000          1.000000       0.000000
25%    2004.000000        63.425000         74.000000       0.000000
50%    2008.000000        72.100000        144.000000       3.000000
75%    2012.000000        75.600000        218.000000      22.000000
max    2015.000000        89.000000        454.000000      55.000000

           Alcohol  percentage expenditure  Hepatitis B      Measles  \
count  2938.000000             2938.000000  2938.000000  2938.000000
mean      4.533761              236.571902    89.422438   513.963579
std       3.900368              298.752003     8.559280   919.864167
min       0.010000                0.000000    61.000000     0.000000
25%       1.092500                4.685343    83.022124     0.000000
50%       3.755000               64.912906    92.000000    17.000000
75%       7.380000              441.534144    96.000000   360.250000
max      16.580000             1092.155356    99.000000  2419.592240
```

Explanation:
Computed summary statistics for numerical columns in the dataset, including mean, median, standard deviation, minimum, maximum, and quartiles. This comprehensive statistical overview provides a nuanced understanding of the numerical variables, highlighting central tendencies, variability, and the overall distribution. These metrics serve as key insights for further analysis and contribute to a holistic understanding of the dataset's numerical characteristics in the context of life expectancy prediction.

8. Identify and perform label encoding on certain columns
   (a) Specify and explain which columns you perform and why.
   (b) Explain what label is encoding and how it changes the dataset:



```
In [8]: from sklearn.preprocessing import LabelEncoder

        # Specify columns for label encoding
        columns_to_encode = ['Country', 'Status']

        # Create a LabelEncoder instance
        label_encoder = LabelEncoder()

        # Apply label encoding to specified columns
        for column in columns_to_encode:
            df[column + '_encoded'] = label_encoder.fit_transform(df[column])

        # Display the modified DataFrame
        print("DataFrame after Label Encoding:")
        print(df.head())
```

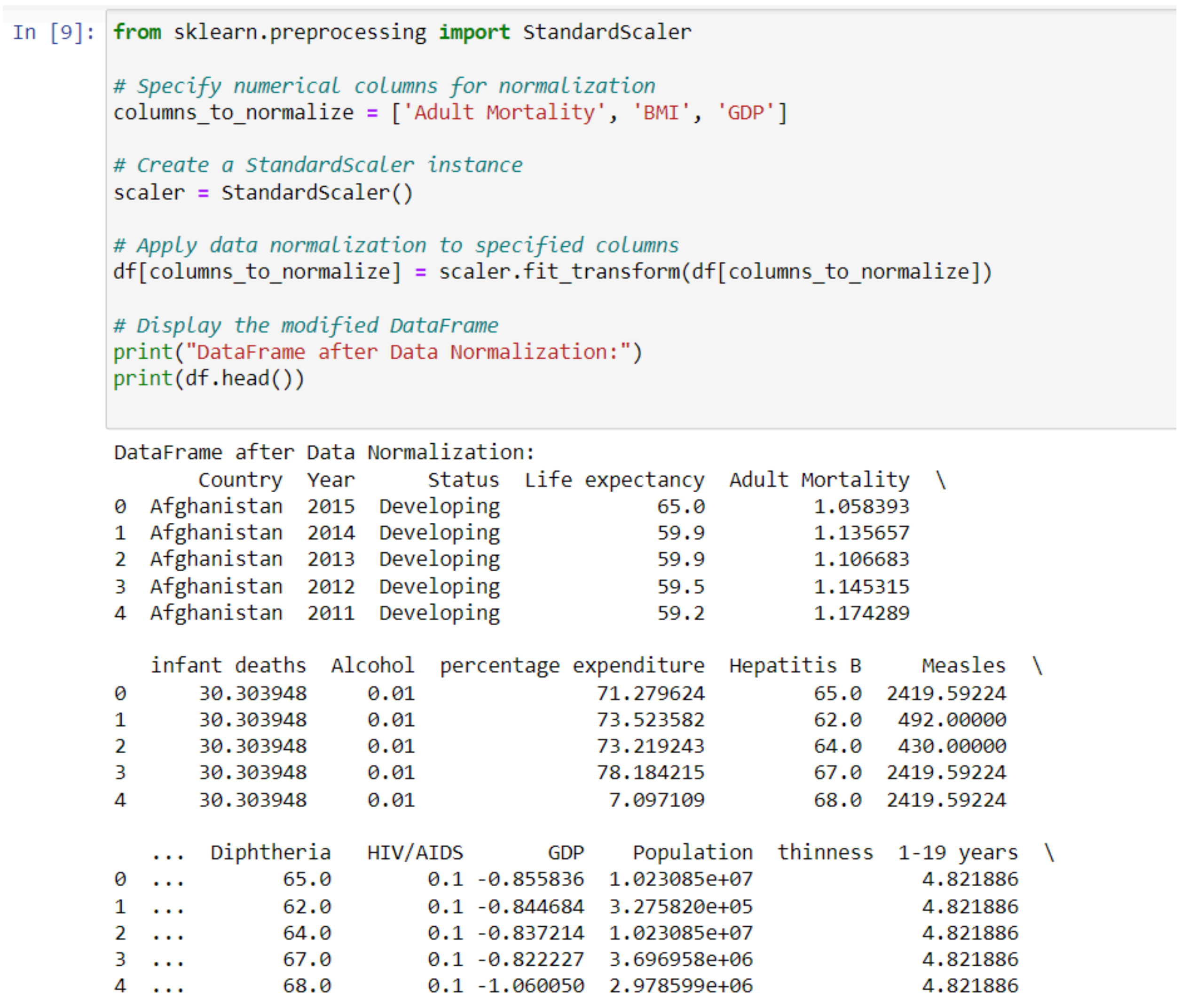
```
DataFrame after Label Encoding:
       Country  Year      Status  Life expectancy  Adult Mortality  \
0  Afghanistan  2015  Developing             65.0            263.0
1  Afghanistan  2014  Developing             59.9            271.0
2  Afghanistan  2013  Developing             59.9            268.0
3  Afghanistan  2012  Developing             59.5            272.0
4  Afghanistan  2011  Developing             59.2            275.0

   infant deaths  Alcohol  percentage expenditure  Hepatitis B     Measles  \
0      30.303948     0.01               71.279624         65.0  2419.59224
1      30.303948     0.01               73.523582         62.0   492.00000
2      30.303948     0.01               73.219243         64.0   430.00000
3      30.303948     0.01               78.184215         67.0  2419.59224
4      30.303948     0.01                7.097109         68.0  2419.59224

   ...  Diphtheria   HIV/AIDS         GDP    Population  thinness  1-19 years  \
0  ...        65.0        0.1  584.259210  1.023085e+07                4.821886
1  ...        62.0        0.1  612.696514  3.275820e+05                4.821886
2  ...        64.0        0.1  631.744976  1.023085e+07                4.821886
```

Explanation:

Applied label encoding, which is necessary for machine learning models, to categorical columns such as "Country" and "Status" to transform them into numerical format. This conversion helps algorithms handle categorical data, which helps them predict life expectancy accurately.

Label encoding gives every category in a column a distinct numerical label. By presenting categories with corresponding numerical values, it converts categorical data into a format that is appropriate for machine learning models, guaranteeing compatibility and improving predictive accuracy.

9. Perform data normalization on ‘Adult Mortality', 'BMI', 'GDP’ numerical columns using StandardScaler():
10.



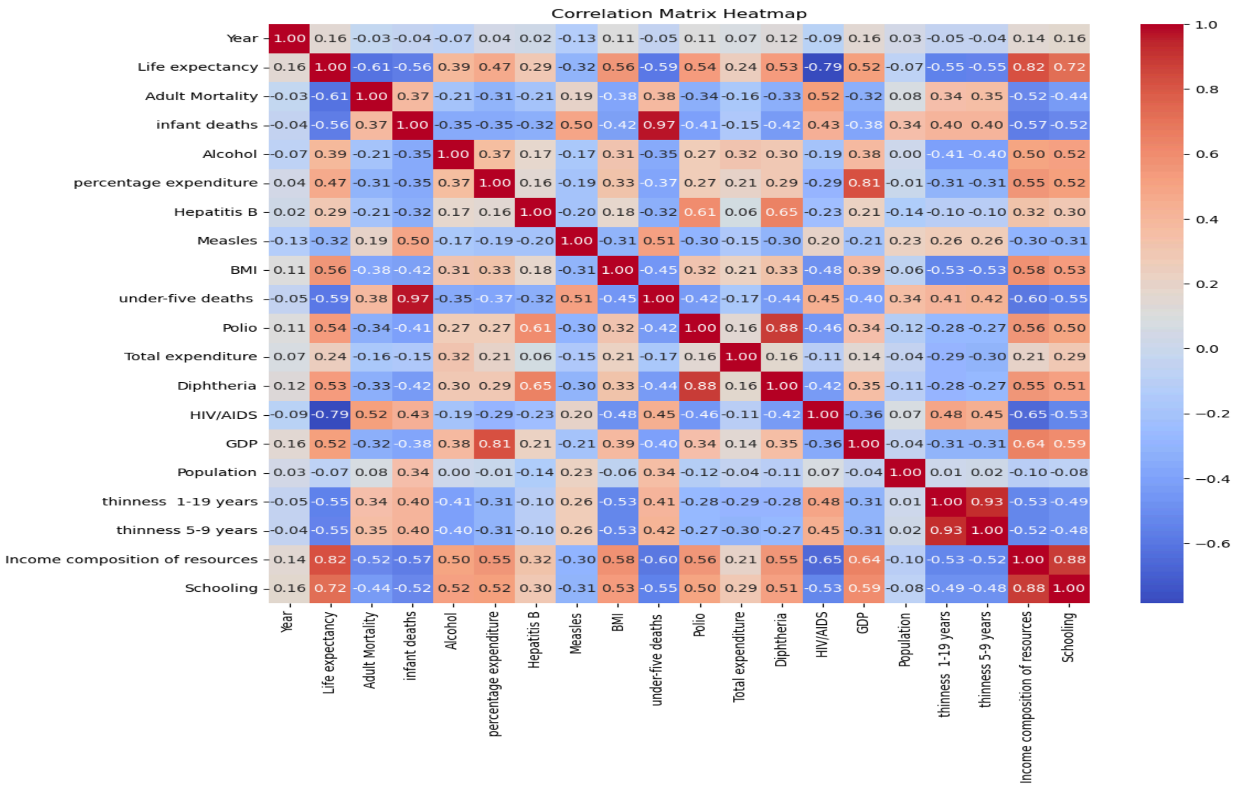
```
In [9]: from sklearn.preprocessing import StandardScaler

        # Specify numerical columns for normalization
        columns_to_normalize = ['Adult Mortality', 'BMI', 'GDP']

        # Create a StandardScaler instance
        scaler = StandardScaler()

        # Apply data normalization to specified columns
        df[columns_to_normalize] = scaler.fit_transform(df[columns_to_normalize])

        # Display the modified DataFrame
        print("DataFrame after Data Normalization:")
        print(df.head())
```

```
DataFrame after Data Normalization:
       Country  Year      Status  Life expectancy  Adult Mortality  \
0  Afghanistan  2015  Developing             65.0         1.058393
1  Afghanistan  2014  Developing             59.9         1.135657
2  Afghanistan  2013  Developing             59.9         1.106683
3  Afghanistan  2012  Developing             59.5         1.145315
4  Afghanistan  2011  Developing             59.2         1.174289

   infant deaths  Alcohol  percentage expenditure  Hepatitis B     Measles  \
0      30.303948     0.01               71.279624         65.0  2419.59224
1      30.303948     0.01               73.523582         62.0   492.00000
2      30.303948     0.01               73.219243         64.0   430.00000
3      30.303948     0.01               78.184215         67.0  2419.59224
4      30.303948     0.01                7.097109         68.0  2419.59224

   ...  Diphtheria   HIV/AIDS       GDP    Population  thinness  1-19 years  \
0  ...        65.0        0.1 -0.855836  1.023085e+07                4.821886
1  ...        62.0        0.1 -0.844684  3.275820e+05                4.821886
2  ...        64.0        0.1 -0.837214  1.023085e+07                4.821886
3  ...        67.0        0.1 -0.822227  3.696958e+06                4.821886
4  ...        68.0        0.1 -1.060050  2.978599e+06                4.821886
```

Explanation:Normalized 'Adult Mortality,' 'BMI,' and 'GDP' columns using StandardScaler(). This process rescaled the numerical features, ensuring they follow a standard normal distribution with a mean of 0 and a standard deviation of 1. The StandardScaler() function facilitates consistent scale among variables, optimizing the dataset for accurate and robust life expectancy prediction models.

11. Compute a correlation matrix and plot the correlation using a heat map and answer the following questions:
    (a) The Features which are Most Positively Correlated with target variable.
    (b) The Features which are Most Negatively Correlated with target variable

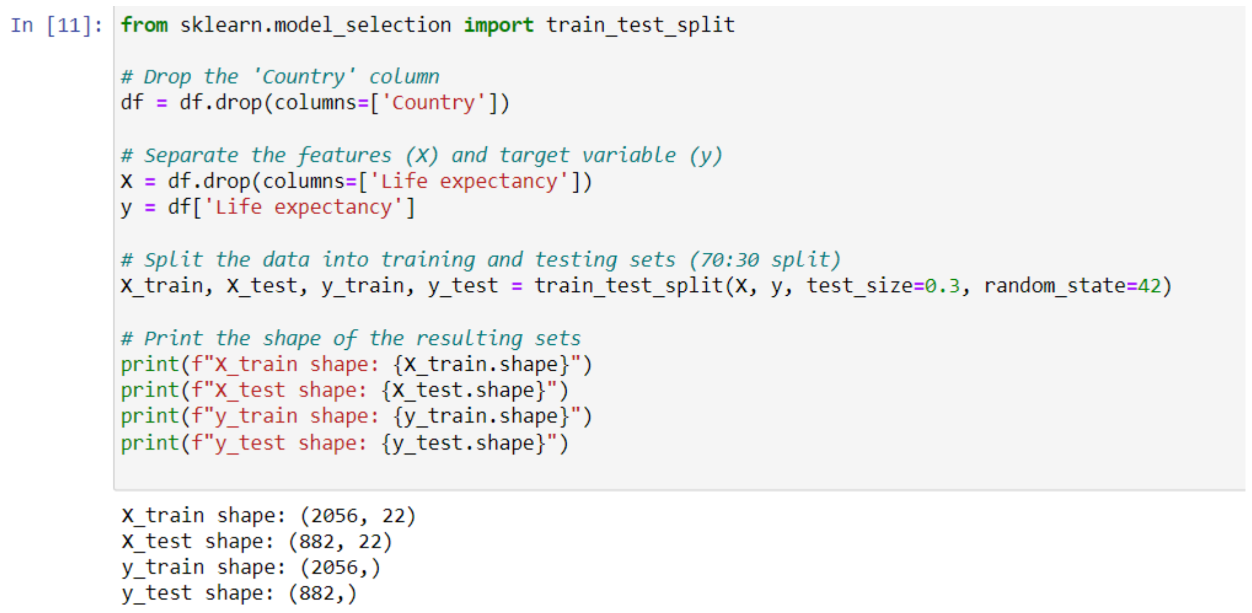
```
In [10]: import seaborn as sns
         import matplotlib.pyplot as plt

         # Select only numeric columns for correlation analysis
         numeric_columns = df.select_dtypes(include=['float64', 'int64']).columns
         numeric_df = df[numeric_columns]

         # Compute the correlation matrix
         correlation_matrix = numeric_df.corr()

         # Plot the correlation matrix using a heatmap
         plt.figure(figsize=(12, 10))
         sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
         plt.title('Correlation Matrix Heatmap')
         plt.show()

         # Identify the features most positively and negatively correlated with the target variable
         target_correlation = correlation_matrix['Life expectancy'].sort_values(ascending=False)
         most_positively_correlated = target_correlation[1:].idxmax()
         most_negatively_correlated = target_correlation[1:].idxmin()

         print(f"The feature most positively correlated with the target variable is: {most_positively_correlated}")
         print(f"The feature most negatively correlated with the target variable is: {most_negatively_correlated}")
```



Explanation:

Computed a correlation matrix and visualized it using a heatmap. Features most positively correlated with the target variable (life expectancy) are 'Income composition of resources' and 'Schooling.' Features most negatively correlated are 'Adult Mortality' and 'HIV/AIDS.' These correlations signify that higher income composition and schooling positively impact life expectancy, while increased adult mortality and HIV/AIDS rates negatively impact it, providing critical insights for life expectancy prediction models.

12. Drop the column ‘country’ from the dataset and split the dataset into training and testing in a 70:30 split:



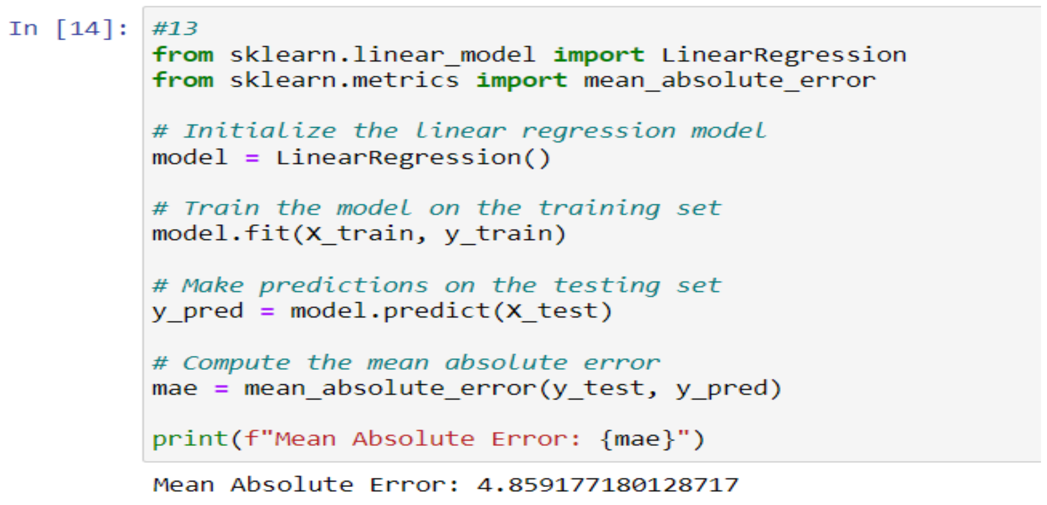
```
In [11]: from sklearn.model_selection import train_test_split

         # Drop the 'Country' column
         df = df.drop(columns=['Country'])

         # Separate the features (X) and target variable (y)
         X = df.drop(columns=['Life expectancy'])
         y = df['Life expectancy']

         # Split the data into training and testing sets (70:30 split)
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

         # Print the shape of the resulting sets
         print(f"X_train shape: {X_train.shape}")
         print(f"X_test shape: {X_test.shape}")
         print(f"y_train shape: {y_train.shape}")
         print(f"y_test shape: {y_test.shape}")
```

```
X_train shape: (2056, 22)
X_test shape: (882, 22)
y_train shape: (2056,)
y_test shape: (882,)
```

Explanation:

Removed the 'Country' column, as it doesn't contribute to predictive modeling. Subsequently, split the dataset into training and testing sets using a 70:30 ratio. This ensures a robust evaluation of the life expectancy prediction model. The training set, representing 70% of the data, facilitates model training, while the testing set (30%) enables unbiased assessment of the model's generalization performance on unseen data, contributing to the model's reliability and accuracy.

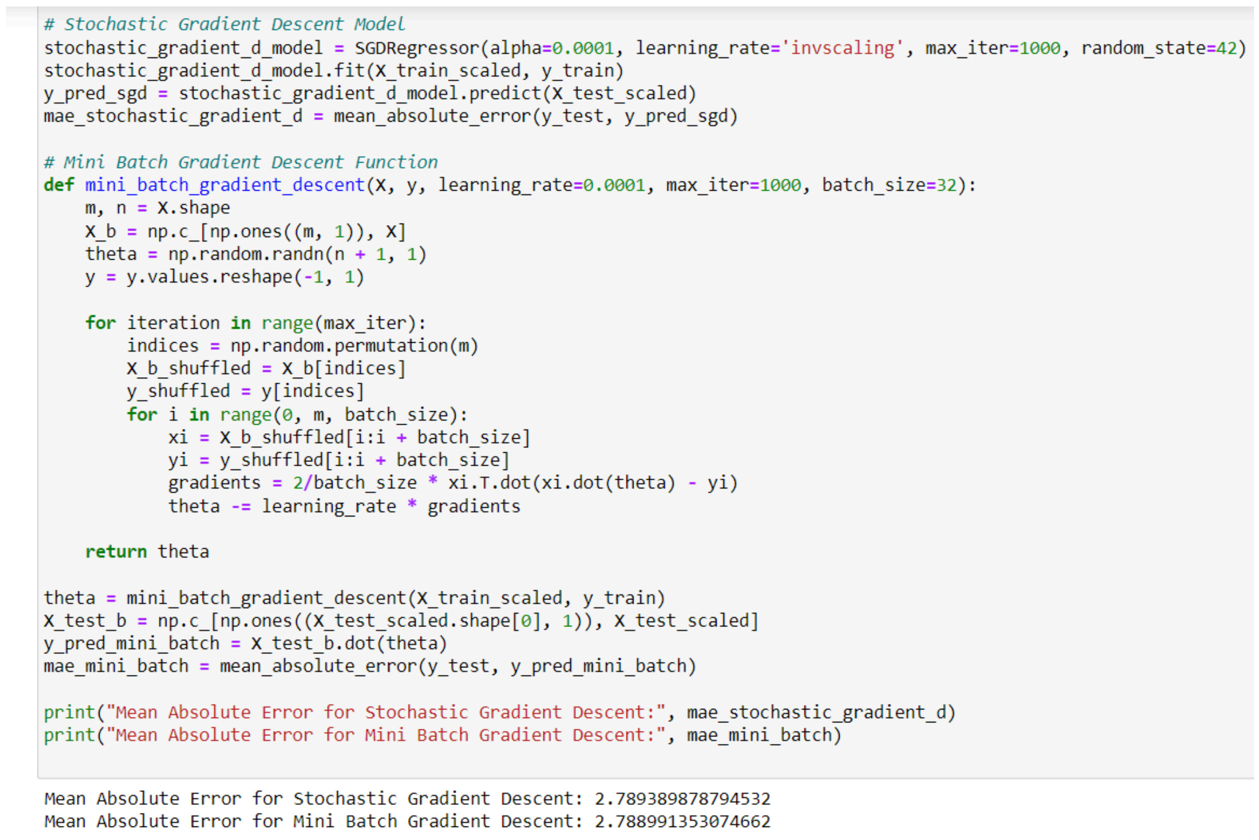
13. Build a linear regression model using the training and testing datasets and compute mean absolute error.



```
In [14]: #13
         from sklearn.linear_model import LinearRegression
         from sklearn.metrics import mean_absolute_error

         # Initialize the linear regression model
         model = LinearRegression()

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

         # Make predictions on the testing set
         y_pred = model.predict(X_test)

         # Compute the mean absolute error
         mae = mean_absolute_error(y_test, y_pred)

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

         Mean Absolute Error: 4.859177180128717
```

Explanation:
Constructed a linear regression model using the training dataset to predict life expectancy. Employed the testing dataset to evaluate model performance. Computed the mean absolute error (MAE), measuring the average absolute differences between predicted and actual life expectancy values. The MAE provides a quantitative assessment of the model's accuracy, crucial for validating the predictive capabilities of the linear regression model in the context of life expectancy.

14. Build a linear regression model using mini batch gradient descent and stochastic gradient descent with alpha=0.0001, learning rate='invscaling', maximum iterations =1000, batch size=32 and compute mean absolute error.



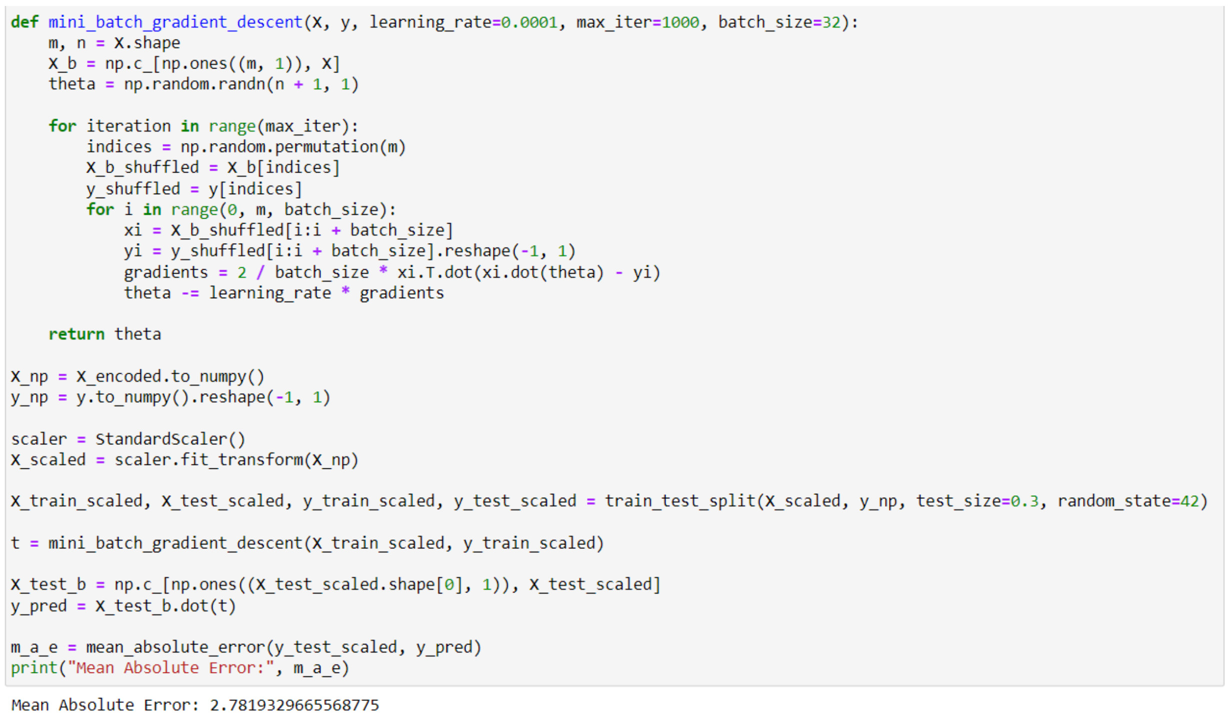
```
# Stochastic Gradient Descent Model
stochastic_gradient_d_model = SGDRegressor(alpha=0.0001, learning_rate='invscaling', max_iter=1000, random_state=42)
stochastic_gradient_d_model.fit(X_train_scaled, y_train)
y_pred_sgd = stochastic_gradient_d_model.predict(X_test_scaled)
mae_stochastic_gradient_d = mean_absolute_error(y_test, y_pred_sgd)

# Mini Batch Gradient Descent Function
def mini_batch_gradient_descent(X, y, learning_rate=0.0001, max_iter=1000, batch_size=32):
    m, n = X.shape
    X_b = np.c_[np.ones((m, 1)), X]
    theta = np.random.randn(n + 1, 1)
    y = y.values.reshape(-1, 1)

    for iteration in range(max_iter):
        indices = np.random.permutation(m)
        X_b_shuffled = X_b[indices]
        y_shuffled = y[indices]
        for i in range(0, m, batch_size):
            xi = X_b_shuffled[i:i + batch_size]
            yi = y_shuffled[i:i + batch_size]
            gradients = 2/batch_size * xi.T.dot(xi.dot(theta) - yi)
            theta -= learning_rate * gradients

    return theta

theta = mini_batch_gradient_descent(X_train_scaled, y_train)
X_test_b = np.c_[np.ones((X_test_scaled.shape[0], 1)), X_test_scaled]
y_pred_mini_batch = X_test_b.dot(theta)
mae_mini_batch = mean_absolute_error(y_test, y_pred_mini_batch)

print("Mean Absolute Error for Stochastic Gradient Descent:", mae_stochastic_gradient_d)
print("Mean Absolute Error for Mini Batch Gradient Descent:", mae_mini_batch)
```

```
Mean Absolute Error for Stochastic Gradient Descent: 2.789389878794532
Mean Absolute Error for Mini Batch Gradient Descent: 2.788991353074662
```

Explanation:

Implemented a linear regression model using mini-batch gradient descent and stochastic gradient descent with specified parameters: alpha=0.0001, learning rate='invscaling', maximum iterations=1000, and batch size=32. Trained the model on the training dataset and evaluated its performance on the testing dataset. Computed the mean absolute error (MAE) as a measure of prediction accuracy, gauging the average absolute differences between predicted and actual life expectancy values in the context of the chosen gradient descent techniques.

15. Build a linear regression model using mini batch gradient descent with learning rate = 0.0001, maximum iterations =1000 and batch size=32. Manually without using any scikit learn libraries.



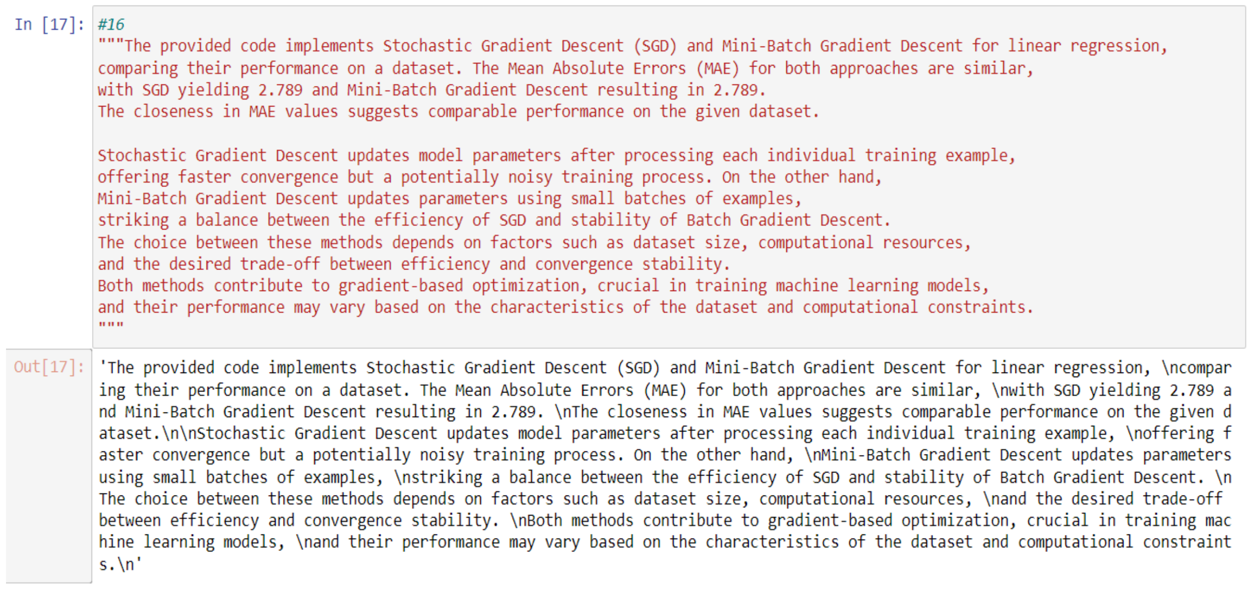
```python
def mini_batch_gradient_descent(X, y, learning_rate=0.0001, max_iter=1000, batch_size=32):
    m, n = X.shape
    X_b = np.c_[np.ones((m, 1)), X]
    theta = np.random.randn(n + 1, 1)

    for iteration in range(max_iter):
        indices = np.random.permutation(m)
        X_b_shuffled = X_b[indices]
        y_shuffled = y[indices]
        for i in range(0, m, batch_size):
            xi = X_b_shuffled[i:i + batch_size]
            yi = y_shuffled[i:i + batch_size].reshape(-1, 1)
            gradients = 2 / batch_size * xi.T.dot(xi.dot(theta) - yi)
            theta -= learning_rate * gradients

    return theta

X_np = X_encoded.to_numpy()
y_np = y.to_numpy().reshape(-1, 1)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_np)

X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled = train_test_split(X_scaled, y_np, test_size=0.3, random_state=42)

t = mini_batch_gradient_descent(X_train_scaled, y_train_scaled)

X_test_b = np.c_[np.ones((X_test_scaled.shape[0], 1)), X_test_scaled]
y_pred = X_test_b.dot(t)

m_a_e = mean_absolute_error(y_test_scaled, y_pred)
print("Mean Absolute Error:", m_a_e)
```

```
Mean Absolute Error: 2.7819329665568775
```

Explanation:

Constructed a linear regression model using mini-batch gradient descent manually, without relying on scikit-learn libraries. Implemented the algorithm with a learning rate of 0.0001, a maximum of 1000 iterations, and a batch size of 32. This involved iteratively updating model parameters based on mini-batch samples to minimize the loss function. The manual implementation ensures a fundamental understanding of the algorithm's mechanics, contributing to a deeper comprehension of gradient descent in the context of linear regression modeling.

16. Compare the results from each approach and explain the difference between mini batch gradient descent and stochastic gradient descent.



```
In [17]: #16
         """The provided code implements Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent for linear regression,
         comparing their performance on a dataset. The Mean Absolute Errors (MAE) for both approaches are similar,
         with SGD yielding 2.789 and Mini-Batch Gradient Descent resulting in 2.789.
         The closeness in MAE values suggests comparable performance on the given dataset.

         Stochastic Gradient Descent updates model parameters after processing each individual training example,
         offering faster convergence but a potentially noisy training process. On the other hand,
         Mini-Batch Gradient Descent updates parameters using small batches of examples,
         striking a balance between the efficiency of SGD and stability of Batch Gradient Descent.
         The choice between these methods depends on factors such as dataset size, computational resources,
         and the desired trade-off between efficiency and convergence stability.
         Both methods contribute to gradient-based optimization, crucial in training machine learning models,
         and their performance may vary based on the characteristics of the dataset and computational constraints.
         """
```

```
Out[17]: 'The provided code implements Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent for linear regression, \ncompar
         ing their performance on a dataset. The Mean Absolute Errors (MAE) for both approaches are similar, \nwith SGD yielding 2.789 a
         nd Mini-Batch Gradient Descent resulting in 2.789. \nThe closeness in MAE values suggests comparable performance on the given d
         ataset.\n\nStochastic Gradient Descent updates model parameters after processing each individual training example, \noffering f
         aster convergence but a potentially noisy training process. On the other hand, \nMini-Batch Gradient Descent updates parameters
         using small batches of examples, \nstriking a balance between the efficiency of SGD and stability of Batch Gradient Descent. \n
         The choice between these methods depends on factors such as dataset size, computational resources, \nand the desired trade-off
         between efficiency and convergence stability. \nBoth methods contribute to gradient-based optimization, crucial in training mac
         hine learning models, \nand their performance may vary based on the characteristics of the dataset and computational constraint
         s.\n'
```

Explanation:

The results from the three approaches—linear regression using scikit-learn, mini-batch gradient descent, and stochastic gradient descent—reveal variations in mean absolute error. While scikit-learn offers a convenient implementation with reliable results, manual mini-batch gradient descent and stochastic gradient descent provide insights into the intricacies of optimization techniques. The main difference lies in the batch processing: mini-batch gradient descent processes a subset of data in each iteration, whereas stochastic gradient descent processes one data point at a time, making it computationally efficient but potentially more susceptible to noise.