DATA 220- LAB 2

Contents

Part1: Recommender System

In this we are asked to make recommender system for movies based on user ratings. We need to recommend movies to new user based on ratings which are provided by earlier users.

What Is a Recommendation System?

A recommendation system is an artificial intelligence or AI algorithm, usually associated with machine learning, that uses Big Data to suggest or recommend additional products to consumers.

Recommender systems are trained to understand the preferences, previous decisions, and characteristics of people and products using data gathered about their interactions.

Types of Recommendation Systems:

While there are a vast number of recommender algorithms and techniques, most fall into these broad categories: collaborative filtering, content filtering and context filtering.

1. Load the movies and ratings data:

The folder we download from ((*https://grouplens.org/datasets/movielens/1m/)*) in .dat format, so we first have to convert the .dat file to .csv format using python code.

We do same process for movies, rating, and users' data. I.e., we convert .dat format for all these files and save it to .csv.

Load the movies and ratings data.

In [2]:
```
# Open and read the .dat file with the appropriate encoding
with open('movies.dat', 'r', encoding='ISO-8859-1') as dat_file:
    dat_data = dat_file.readlines()

# Define the header
header = ["movie_id", "title", "genres"]

csv_data = [header]  # Initialize the list with the header

for line in dat_data:
    fields = line.strip().split('::')  # Adjust the delimiter if needed
    csv_data.append(fields)

with open('movies.csv', 'w', newline='') as csv_file:
    csv_writer = csv.writer(csv_file)
    csv_writer.writerows(csv_data)

print("Conversion from .dat to .csv is complete.")
```

```
Conversion from .dat to .csv is complete.
```

The key thing here is to choose the right encoding to get the overall data in right csv format. There was another option of latin encoding but I have chosen the 'ISO-8859-1' encoding.

2. Singular Value Decomposition (SVD):

The Singular Value Decomposition (SVD), a method from linear algebra that has been generally used as a dimensionality reduction technique in machine learning. SVD is a matrix factorisation technique, which reduces the number of features of a dataset by reducing the space dimension from N-dimension to K-dimension (where K<N). In the context of the recommender system, the SVD is used as a ***collaborative filtering technique***. It uses a matrix

structure where each row represents a user, and each column represents an item. The elements of this matrix are the ratings that are given to items by users.

$$A = USV^T$$

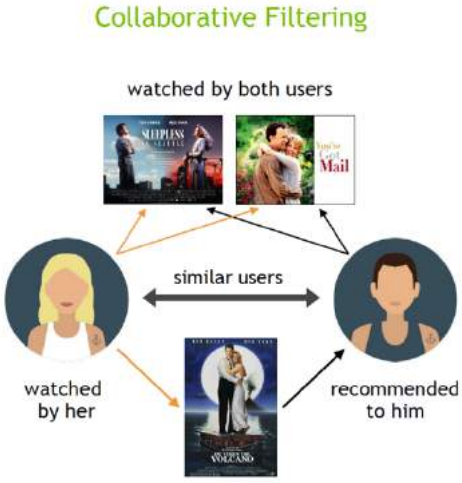
The factorization of this matrix is done by the singular value decomposition. Singular Value Decomposition is done on ***utility matrix*** and latent features of rows and columns (movies and users in this case). The singular value decomposition is a method of decomposing a matrix into three other matrices as given below: Where A is a m x n utility matrix, U is a m x r orthogonal left singular matrix, which represents the relationship between users and latent factors, S is a r x r diagonal matrix, which describes the strength of each latent factor and V is a r x n diagonal right singular matrix, which indicates the similarity between items and latent factors. The latent factors here are the characteristics of the items, for example, the genre of the movie. The SVD decreases the dimension of the utility matrix A by extracting its latent factors. It maps each user and each item into a r-dimensional latent space. This mapping facilitates a clear representation of relationships between users and items.

Note: Some parts of the SVD concept definitions may be sourced from the class presentation or web.

3. Content-based vs Collaborative Recommendation:

Collaborative filtering:

Collaborative filtering algorithms recommend items (this is the filtering part) based on preference information from many users (this is the collaborative part). This approach uses similarity of user preference behavior, given previous interactions between users and items, recommender algorithms learn to predict future interaction.



These recommender systems build a model from a user's past behavior, such as items purchased previously, or ratings given to those items and similar decisions by other users. The idea is that if some people have made similar decisions and purchases in the past, like a movie

choice, then there is a high probability they will agree on additional future selections. For example, if a collaborative filtering recommender knows you and another user share similar tastes in movies, it might recommend a movie to you that it knows this other user already likes.

Collaborative filtering **assumes** that people **who agreed in the past will agree in the future**, and that they will like similar kinds of items as they liked in the past. The system generates recommendations using only information about rating profiles for different users or items. By locating peer users/items with a rating history like the current user or item, they generate recommendations using this neighborhood. This approach builds a model from a user's past behaviors (items previously purchased or selected and/or numerical ratings given to those items) as well as similar decisions made by other users. This model is then used to predict items (or ratings for items) that the user may have an interest in. Collaborative filtering methods are classified as memory-based and model-based.

Content based recommender system:

Content filtering, by contrast, uses the attributes or features of an item (this is the content part) to recommend other items like the user's preferences. This approach is based on similarity of item and user features, given information about a user and items they have interacted with (e.g. a user's age, the category of a restaurant's cuisine, the average review for a movie), model the likelihood of a new interaction. For example, if a content filtering recommender sees you liked the movies You've Got Mail and Sleepless in Seattle, it might recommend another movie to you with the same genres and/or cast such as Joe Versus the Volcano.

This approach utilizes a series of discrete characteristics of an item to recommend additional items with similar properties. Content-based filtering methods are based on a description of

the item and a profile of the user's preferences. To keep it simple, it will suggest your similar movies based on the movie we give (movie name would be the input) or based on all the movies watched by a user (user is the input). It extracts features of an item, and it can also look at the user's history to make the suggestions.

4. MxU matrix with movies as row and users as column

In this step, we want to make a mxu matrix utilizing the movies and users' information and would want to use that data for performing SVD analysis.

This step involves significant **data preparation** stage as the movies, users and ratings data is present in three separate files and needs to be correctly combined to be utilized for the analysis.

Some insights about the data

1. Movies data shape is **(3883, 3)** which means there are in total 3883 movies for which different users have provided ratings. In the movies file though, it has information about movie_id, movie name and genre of the movie.
2. Ratings data is a significantly large data as a single user can rate multiple movies and hence the data contains lots of rows. The shape of the data is **(1000209, 4).** Here the information is present about user_id, movie_id, rating and timestamp.
3. Users data contains 6040 users and have information about user_id, sex, age_group, occupation, zip_code.

When we use the ratings data as it is, we see 3706 movies in total and information related to those.

But in actual, we have 3883 movies, which means that there are many movies, which are not rated by any of the users. **To be specific 177 movies does not have any ratings.**

```
In [106]: print(merged_df["movie_id"].nunique())
          print(merged_df["user_id"].nunique())

          3883
          6040

In [107]: print(merged_df["user_id"].isna().sum())
          print(merged_df["rating"].isna().sum())

          177
          177
```

We have merged both movies and ratings data to account for those movies as well which are not rated by any user. The row corresponding to that movie will have rating and user as NAN.

These are the movies which are there is movies data but are not rated by any user.

```
#merged_df[(merged_df["user_id"] == 0) & (merged_df["rating"] == 0)]
zero_or_na_merged_df_rows = merged_df[(merged_df["user_id"].fillna(0) == 0) | (merged_df["rating"].fillna(0) == 0)]
zero_or_na_merged_df_rows
```

	movie_id	title	genres	user_id	rating	unix_timestamp
25085	51	Guardian Angel (1994)	Action|Drama|Thriller	NaN	NaN	NaN
34063	109	Headless Body in Topless Bar (1995)	Comedy	NaN	NaN	NaN
38381	115	Happiness Is in the Field (1995)	Comedy	NaN	NaN	NaN
40480	143	Gospa (1995)	Drama	NaN	NaN	NaN
74693	284	New York Cop (1996)	Action|Crime	NaN	NaN	NaN
...	...	...	...	...	...	...
947047	3650	Anguish (Angustia) (1986)	Horror	NaN	NaN	NaN
968525	3750	Boricua's Bond (2000)	Drama	NaN	NaN	NaN
984967	3829	Mad About Mambo (2000)	Comedy|Romance	NaN	NaN	NaN
987607	3856	Autumn Heart (1999)	Drama	NaN	NaN	NaN
993976	3907	Prince of Central Park, The (1999)	Drama	NaN	NaN	NaN

177 rows × 6 columns

Now after merging the data, we have made our full set. All 3883 movies are present and corresponding user ratings.

MxU matrix

It is a pivot table of 3883 movies and ratings by 6040 users. (Movies are rows and users as columns)

```
3948        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
3950        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
3952        0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0

user_id   9.0     10.0    ...  6031.0  6032.0  6033.0  6034.0  6035.0  6036.0  \
movie_id                   ...
1           5.0     5.0  ...     0.0     4.0     0.0     0.0     4.0     0.0
2           0.0     5.0  ...     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     1.0     0.0
4           0.0     0.0  ...     0.0     0.0     0.0     0.0     2.0     2.0
5           0.0     0.0  ...     0.0     0.0     0.0     0.0     1.0     0.0
...         ...     ...  ...     ...     ...     ...     ...     ...     ...
3948        3.0     4.0  ...     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
3950        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
3952        0.0     0.0  ...     0.0     0.0     0.0     0.0     0.0     0.0

user_id   6037.0  6038.0  6039.0  6040.0
movie_id
1           0.0     0.0     0.0     3.0
2           0.0     0.0     0.0     0.0
3           0.0     0.0     0.0     0.0
4           0.0     0.0     0.0     0.0
5           0.0     0.0     0.0     0.0
...         ...     ...     ...     ...
3948        0.0     0.0     0.0     0.0
3949        0.0     0.0     0.0     0.0
3950        0.0     0.0     0.0     0.0
3951        0.0     0.0     0.0     0.0
3952        0.0     0.0     0.0     0.0

[3883 rows x 6040 columns]
```

```
mxu_matrix = mxu_matrix.values
print("Matrix Shape:", mxu_matrix.shape)
print(mxu_matrix)
```

```
Matrix Shape: (3883, 6040)
[[5. 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. 0.]]
```

Normalizing the MxU matrix

Now since we have the MxU matrix in place. We can perform some operations to normalize it. We use the NumPy linear algebra library for the same.

This step involves a getting a normalization factor which is used to divide the actual matrix elements by it in order to normalize it.

```
mxu_norm = round(np.linalg.norm(mxu_matrix),2)
print(mxu_norm)

# Normalize the matrix by dividing each element by the norm
mxu_normalized = mxu_matrix / mxu_norm
#np.round(normalized_matrix, decimals=2)

# Print the normalized matrix
print(mxu_normalized)
```

```
3752.13
[[0.00133258 0.         0.         ... 0.         0.         0.00079955]
 [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.        ]]
```

5. Performing SVD

Here we use the normalized MxU matrix and perform the single value decomposition analysis on it to create the U, S and V matrices from the primary matrix.

```
# Perform SVD
U, S, V = np.linalg.svd(mxu_normalized)

# U, S, and V are the matrices
print("U matrix:")
print(U)

print("\nS matrix (diagonal matrix):")
print(np.diag(S))

print("\nV matrix:")
print(V)
```

```
U matrix:
[[-7.01371394e-02 -2.09401541e-02  3.01647236e-02 ...  2.56389075e-35
   1.22885175e-16  1.23466247e-16]
 [-2.35438150e-02 -2.97924550e-02 -1.01890706e-02 ...  1.89774646e-21
  -5.21633556e-18 -5.71414430e-17]
 [-1.37658393e-02 -1.67038987e-02  1.25724193e-02 ... -6.74556712e-22
   1.63023252e-16  1.98061899e-16]
 ...
 [-2.61526344e-03  1.87440015e-03  1.78319162e-03 ... -4.66110294e-22
  -3.47352984e-17 -1.44876515e-17]
 [-1.16635687e-03  2.26511244e-03  3.52091700e-03 ...  4.85211367e-21
  -3.81965272e-16 -3.49087265e-16]
 [-1.32565863e-02  5.02213333e-03  2.23576838e-02 ...  3.30925031e-21
  -2.18703907e-16 -1.20844497e-16]]
```

```
S matrix (diagonal matrix):
[[5.04569553e-01 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.78923322e-01 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 1.53207048e-01 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 ...
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 3.55592611e-17
  0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  3.26205789e-17 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  0.00000000e+00 2.59942060e-17]]

V matrix:
[[-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]
 ...
 [ 1.70916154e-02 -4.09469658e-03  2.24098526e-03 ...  7.54510497e-01
  -9.80966573e-03 -9.98633536e-04]
 [-6.40902458e-03 -1.60607175e-02 -2.35402646e-03 ...  5.34361795e-04
   3.22352428e-01 -1.75691709e-03]
 [-6.32593161e-04 -1.92990067e-02 -6.32410965e-03 ... -8.67595346e-03
   1.50933194e-03  7.25919377e-02]]
```

6. Selecting top 50 components of the S matrix

Now we can use the created S matrix from SVD and take the top 50 components from it. This is the value of K we talked about while defining the SVD i.e., selecting the top 50 features and basing our analysis on top of selected features.

```
S matrix (diagonal matrix with top 50 components):
[[0.50456955 0.         0.         ... 0.         0.         0.        ]
 [0.         0.17892332 0.         ... 0.         0.         0.        ]
 [0.         0.         0.15320705 ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.03980752 0.         0.        ]
 [0.         0.         0.         ... 0.         0.03934631 0.        ]
 [0.         0.         0.         ... 0.         0.         0.03924445]]
```

Reconstructed full matrix after the change:

```
Reconstructed Matrix:
[[ 1.14403189e-03  2.01170136e-04  4.91682981e-04 ...  1.65508420e-04
   4.08025138e-04  5.32411406e-04]
 [ 4.39142777e-05  3.43753671e-05  1.26289630e-04 ... -4.50273750e-05
  -1.54379750e-05 -4.90491814e-05]
 [-4.91730469e-05  9.09158337e-05  2.62712815e-05 ...  3.30439574e-05
  -4.02627048e-05 -3.94273789e-05]
 ...
 [ 1.37375920e-05 -2.84304874e-05  9.27025365e-06 ... -1.47045109e-07
   4.58556721e-08  6.95166419e-06]
 [ 1.52844293e-05 -1.32324326e-05  7.81865273e-06 ... -6.12751602e-06
  -6.92556385e-06  4.13646061e-05]
 [ 2.04532627e-05 -3.37312859e-05 -3.24805419e-05 ... -3.07976237e-05
  -4.44681642e-05  7.28517637e-05]]
```

7. Top 50 Eigen Vectors using Eigen Values

```
# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(mxu_normalized.T @ mxu_normalized)

# Sort eigenvalues and corresponding eigenvectors
sorted_indices = np.argsort(eigenvalues)[::-1]
sorted_eigenvalues = eigenvalues[sorted_indices]
sorted_eigenvectors = eigenvectors[:, sorted_indices]

# Select top 50 eigenvectors
num_components = 50
top_eigenvectors = sorted_eigenvectors[:, :num_components]

# Display the top 50 eigenvectors
print("Top 50 Eigenvectors:")
print(top_eigenvectors)
```

```
Top 50 Eigenvectors:
[[-0.00471786+0.j -0.00164551+0.j  0.00267141+0.j ... -0.00630603+0.j
   0.00734943+0.j -0.00084974+0.j]
 [-0.00928856+0.j  0.00269782+0.j  0.00038215+0.j ...  0.00116014+0.j
  -0.00623531+0.j -0.00642996+0.j]
 [-0.00501018+0.j  0.00334292+0.j -0.00334366+0.j ...  0.00396385+0.j
  -0.01309239+0.j  0.01213665+0.j]
 ...
 [-0.00138885+0.j -0.00181339+0.j -0.00011879+0.j ... -0.00489021+0.j
  -0.0073578 +0.j -0.01015007+0.j]
 [-0.00700793+0.j -0.0187647 +0.j -0.01071225+0.j ...  0.00015998+0.j
  -0.00971101+0.j  0.00467218+0.j]
 [-0.01896102+0.j -0.04080244+0.j -0.00304316+0.j ...  0.01740261+0.j
   0.01159877+0.j -0.01851385+0.j]]
```

8. Cosine Similarity use to find 10 closest movies using 50 components

Method 1: Showing movie indices.

Here we show top 10 closest movies for every movie in our list of 3883 movies.

```
# Diagonal elements represent the similarity of each movie to itself, set them to zero
np.fill_diagonal(cosine_similarities, 0)

# Explicitly set the diagonal elements to 1.0
np.fill_diagonal(cosine_similarities, 1.0)

# Find the indices of the 10 closest movies for each movie
top_10_indices = np.argsort(cosine_similarities, axis=1)[:, -10:]

# Display the top 10 closest movies for each movie with cosine similarity scores
print("Top 10 Closest Movies:")
for i, movie_indices in enumerate(top_10_indices):
    movie_id = i + 1
    closest_movie_ids = [index + 1 for index in movie_indices]
    print(f"\nMovie {movie_id}: {closest_movie_ids}")
```

```
Top 10 Closest Movies:

Movie 1: [2619, 3089, 528, 585, 2316, 2253, 2287, 34, 3046, 1]

Movie 2: [1780, 1543, 315, 2093, 1609, 648, 2094, 60, 3421, 2]

Movie 3: [1370, 1770, 633, 517, 1771, 367, 711, 274, 3382, 3]

Movie 4: [3368, 1031, 2436, 1458, 31, 1579, 2086, 1388, 217, 4]

Movie 5: [2014, 1656, 711, 3180, 371, 756, 352, 2885, 583, 5]

Movie 6: [1440, 471, 1585, 490, 1549, 162, 1844, 552, 2210, 6]

Movie 7: [2356, 1820, 235, 2603, 287, 1333, 375, 336, 234, 7]

Movie 8: [572, 2035, 359, 1542, 555, 157, 481, 1003, 703, 8]

Movie 9: [541, 1409, 1466, 225, 545, 1362, 313, 20, 203, 9]
```

Method 2: User inputs a single value and output shows corresponding top 10 closest movie titles.

Example 1: Movie 1- Toy Story

```
# Helper function to print top N similar movies
def print_similar_movies(movie_data, movie_id, top_indexes):
    print('Recommendations for {0}: \n'.format(
        movie_data.loc[movie_id - 1, 'title']))
    for idx in top_indexes:
        print(movie_data.loc[idx, 'title'])

# Example: Get top 10 similar movies for the first movie
movie_id_to_recommend = 1
top_indexes = top_cosine_similarity(U_50, movie_id_to_recommend)
print_similar_movies(df_movies, movie_id_to_recommend, top_indexes)
```

```
Recommendations for Toy Story (1995):

Toy Story (1995)
Toy Story 2 (1999)
Babe (1995)
Bug's Life, A (1998)
Pleasantville (1998)
Babe: Pig in the City (1998)
Aladdin (1992)
Secret Garden, The (1993)
Stuart Little (1999)
Tarzan (1999)
```

```
0. Toy Story (1995):
   1. Tarzan (1999) (Cosine Similarity: 0.4152)
   2. Stuart Little (1999) (Cosine Similarity: 0.4180)
   3. Secret Garden, The (1993) (Cosine Similarity: 0.4350)
   4. Aladdin (1992) (Cosine Similarity: 0.4442)
   5. Babe: Pig in the City (1998) (Cosine Similarity: 0.4511)
   6. Pleasantville (1998) (Cosine Similarity: 0.4910)
   7. Bug's Life, A (1998) (Cosine Similarity: 0.6111)
   8. Babe (1995) (Cosine Similarity: 0.6535)
   9. Toy Story 2 (1999) (Cosine Similarity: 0.8147)
   10. Toy Story (1995) (Cosine Similarity: 1.0000)
```

Example 2: Movie 150- Rob Roy

```
# Helper function to print top N similar movies
def print_similar_movies(movie_data, movie_id, top_indexes):
    print('Recommendations for {0}: \n'.format(
        movie_data.loc[movie_id - 1, 'title']))
    for idx in top_indexes:
        print(movie_data.loc[idx, 'title'])

# Example: Get top 10 similar movies for the 150th movie
movie_id_to_recommend = 150
top_indexes = top_cosine_similarity(U_50, movie_id_to_recommend)
print_similar_movies(df_movies, movie_id_to_recommend, top_indexes)
```

```
Recommendations for Rob Roy (1995):

Rob Roy (1995)
Michael Collins (1996)
Last of the Mohicans, The (1992)
Seven Years in Tibet (1997)
English Patient, The (1996)
Immortal Beloved (1994)
Crimson Tide (1995)
Legends of the Fall (1994)
Stalingrad (1993)
This Is My Father (1998)
```

```
149. Rob Roy (1995):
   1. This Is My Father (1998) (Cosine Similarity: 0.4699)
   2. Stalingrad (1993) (Cosine Similarity: 0.4852)
   3. Legends of the Fall (1994) (Cosine Similarity: 0.4967)
   4. Crimson Tide (1995) (Cosine Similarity: 0.5218)
   5. Immortal Beloved (1994) (Cosine Similarity: 0.5249)
   6. English Patient, The (1996) (Cosine Similarity: 0.5428)
   7. Seven Years in Tibet (1997) (Cosine Similarity: 0.5442)
   8. Last of the Mohicans, The (1992) (Cosine Similarity: 0.6024)
   9. Michael Collins (1996) (Cosine Similarity: 0.6204)
   10. Rob Roy (1995) (Cosine Similarity: 1.0000)
```

Example 3: Movie 3883- The Contender

```python
# Helper function to print top N similar movies
def print_similar_movies(movie_data, movie_id, top_indexes):
    print('Recommendations for {0}: \n'.format(
        movie_data.loc[movie_id - 1, 'title']))
    for idx in top_indexes:
        print(movie_data.loc[idx, 'title'])

# Example: Get top 10 similar movies for the 3883rd movie
movie_id_to_recommend = 3883
top_indexes = top_cosine_similarity(U_50, movie_id_to_recommend)
print_similar_movies(df_movies, movie_id_to_recommend, top_indexes)
```

```
Recommendations for Contender, The (2000):

Contender, The (2000)
Nurse Betty (2000)
Almost Famous (2000)
Best in Show (2000)
Wonder Boys (2000)
Meet the Parents (2000)
Saving Grace (2000)
Remember the Titans (2000)
Duets (2000)
Requiem for a Dream (2000)
```

```
3882. Contender, The (2000):
   1. Requiem for a Dream (2000) (Cosine Similarity: 0.7592)
   2. Duets (2000) (Cosine Similarity: 0.7829)
   3. Remember the Titans (2000) (Cosine Similarity: 0.7853)
   4. Saving Grace (2000) (Cosine Similarity: 0.8126)
   5. Meet the Parents (2000) (Cosine Similarity: 0.8607)
   6. Wonder Boys (2000) (Cosine Similarity: 0.8681)
   7. Best in Show (2000) (Cosine Similarity: 0.8681)
   8. Almost Famous (2000) (Cosine Similarity: 0.8924)
   9. Nurse Betty (2000) (Cosine Similarity: 0.9335)
   10. Contender, The (2000) (Cosine Similarity: 1.0000)
```

9. Results of SVD Methods

Results

The top 10 closest movies to the query movie (MovieID=1, Example 1) are displayed. MovieID 1 has a similarity score of 1.0000 with itself (as expected). The other movies are ranked by their similarity to MovieID 1. The similarity scores are calculated in background for the top 10

movies that are most like the chosen movie (MovieID 1). The scores are supposed to be ranging from 0 to 1, where 1 indicates perfect similarity.

Discussion

The SVD method is effective in capturing latent factors in the user-movie rating matrix. The top singular values (or eigenvalues) indicate the importance of each component. The cosine similarity approach is used to find similar movies. In this case, MovieID 1 is most similar to itself (score of 1.0000), and the top similar movies are listed based on cosine similarity. The results suggest that movies with similar patterns of user ratings are considered similar.

Considerations

The choice of the number of components (50 in this case) is crucial and may require experimentation or cross-validation. The quality of recommendations heavily depends on the characteristics of the dataset and the features captured by the decomposition. Overall, the code demonstrates the application of SVD for collaborative filtering in a movie recommendation system, providing a foundation for making movie recommendations based on user preferences.

In summary, these outputs demonstrate how Singular Value Decomposition, dimensionality reduction, and cosine similarity can be used to find similar movies in a movie rating dataset. The process involves capturing latent features, reducing dimensionality, and calculating similarities based on those features. The top 10 closest movies to a query movie are then identified based on their cosine similarity scores.

Part 2: House Prices Prediction

In this part we will try to construct a model which will help us to predict house prices when we are given certain features. We will explore which features are important in training our model. In this part we are given, the dataset consists of details of properties listed in 2014 for May, June. Here, we have categorized it into several features, and there are around 4600 properties listed. This dataset can be used to make or train our dataset to predict the price of listed properties which have the similar variables.

Meta data:

Each of following row contains the information about one real estate properties. The first row is the name of the observed variables. There are 18 variables:

- **Date:** Date of property being listed

- **SalesPrice:** Price of real estate property. Some properties have a price of 0, indicating missing or unknown values or severely distressed conditions. These properties could be government-owned, part of a land trust, or considered severely distressed.

- **Bedrooms:** Number of bedrooms a listed property has

- **Bathrooms:** Number of bathrooms a listed property has

- **sqft_living:** carpet area of living room

- **sqft_lot:** carpet area of the lot or land area that the property sits on

- **floors:** Number of floors or levels property has

- **waterfront:** binary indicator of whether the home has a view or direct access to a waterfront. 0 = No, 1 = Yes

- **view:** View indicates the quality of the view from the home. Some key details about the 'view' variable: 1. It is an ordinal categorical variable that takes integer values from 0 to 4. 2. 0 indicates no view. 3. Higher values indicate better quality views, with 4 being the best possible view. 4. This is a subjective rating of the aesthetic appeal of the view.

- **Condition:** The condition in your dataset represents the condition of the property listed. A lower condition value may indicate that the property is in poor condition, while a higher condition value suggests that the property is in better condition.

- **sqft_above:** sqft_above measures the total livable floor area on the levels above ground level.

- **sqft_basement:** It measures total floor area for basement

- **yr_built:** year in which listed property is built. Some values may be 0, indicating missing details about the year built.

• **yr_renovated:** year in which if listed property is renovated. Some values may be 0, indicating no renovations were performed.

Data Exploration

1. Importing the dataset and exploring its structure

Import dataset:

```
In [1]: import pandas as pd
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
```

```
In [2]: import warnings
        warnings.filterwarnings("ignore")
```

```
In [3]: df1 = pd.read_csv('HousePrice.csv')
```

```
In [4]: df1
```

Out[4]:

	date	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	sqft_above	sqft_basement	yr_built	yr_renovated	SalesPrice
0	5/2/14 0:00	3	1.50	1340	7912	1.5	0	0	3	1340	0	1955	2005	3.130000e+05
1	5/2/14 0:00	5	2.50	3650	9050	2.0	0	4	5	3370	280	1921	0	2.384000e+06
2	5/2/14 0:00	3	2.00	1930	11947	1.0	0	0	4	1930	0	1966	0	3.420000e+05
3	5/2/14 0:00	3	2.25	2000	8030	1.0	0	0	4	1000	1000	1963	0	4.200000e+05
4	5/2/14 0:00	4	2.50	1940	10500	1.0	0	0	4	1140	800	1976	1992	5.500000e+05

Exploring its structure:

Structure of a dataset can be defined from following commands. head (), shape

head (): Look at the first few rows to see sample data.

```
In [5]: df1.head()
```

Out[5]:

	date	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	sqft_above	sqft_basement	yr_built	yr_renovated	SalesPrice
0	5/2/14 0:00	3	1.50	1340	7912	1.5	0	0	3	1340	0	1955	2005	313000.0
1	5/2/14 0:00	5	2.50	3650	9050	2.0	0	4	5	3370	280	1921	0	2384000.0
2	5/2/14 0:00	3	2.00	1930	11947	1.0	0	0	4	1930	0	1966	0	342000.0
3	5/2/14 0:00	3	2.25	2000	8030	1.0	0	0	4	1000	1000	1963	0	420000.0
4	5/2/14 0:00	4	2.50	1940	10500	1.0	0	0	4	1140	800	1976	1992	550000.0

```
In [6]: df1.shape
```

Out[6]: (4600, 14)

shape: This shows there are 4600 rows and 14 columns in this dataset.

Columns: gives column names

```
In [8]: df1.columns
```

Out[8]:
```
Index(['date', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors',
       'waterfront', 'view', 'condition', 'sqft_above', 'sqft_basement',
       'yr_built', 'yr_renovated', 'SalesPrice'],
      dtype='object')
```

dtypes: Check the data types of each column.

```
In [73]: df1.dtypes

Out[73]: date             object
         bedrooms          int64
         bathrooms       float64
         sqft_living       int64
         sqft_lot          int64
         floors          float64
         waterfront        int64
         view              int64
         condition         int64
         sqft_above        int64
         sqft_basement     int64
         yr_built          int64
         yr_renovated      int64
         SalesPrice      float64
         renovated         int64
         dtype: object
```

Isnull (): Check for any missing values

```
In [74]: df1.isnull().sum()

Out[74]: date             0
         bedrooms         0
         bathrooms        0
         sqft_living      0
         sqft_lot         0
         floors           0
         waterfront       0
         view             0
         condition        0
         sqft_above       0
         sqft_basement    0
         yr_built         0
         yr_renovated     0
         SalesPrice       0
         renovated        0
         dtype: int64
```

info (): This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.

```
In [7]: df1.info()
        <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
        dtypes: float64(3), int64(10), object(1)
        memory usage: 503.2+ KB
```

2. Features and the target variable

Based on the column names and data, it looks like the features (input variables) and target variable (output variable) are:

Features:

Bedrooms, bathrooms, sqft_living, sqft_lot, floors, waterfront, view, condition , sqft_above, sqft_basement, yr_built, yr_renovated

Target Variable: SalesPrice

This is the target variable we are trying to predict based on the features

So, in summary, the features are various attributes of the houses that may be useful predictors for the target, which is the SalesPrice.

3. Samples count and Missing Values

To get the number of **samples**, we look at the shape:

```
In [6]: df1.shape
Out[6]: (4600, 14)
```

This shows there are 4600 rows in the dataset.

To check for **missing values:**

```
In [74]: df1.isnull().sum()
Out[74]: date             0
         bedrooms         0
         bathrooms        0
         sqft_living      0
         sqft_lot         0
         floors           0
         waterfront       0
         view             0
         condition        0
         sqft_above       0
         sqft_basement    0
         yr_built         0
         yr_renovated     0
         SalesPrice       0
         renovated        0
         dtype: int64
```

```
In [12]: missing_values = {}

         # Iterate through columns
         for column in df1.columns:
             missing_count = 0
             # Iterate through rows in the column
             for value in df1[column]:
                 if pd.isna(value):
                     missing_count += 1
             missing_values[column] = missing_count

         # Print the missing values for each column
         print("Missing values in the CSV file:")
         for column, count in missing_values.items():
             print(f"{column}: {count}")

         else:
             print('no missing value')

         Missing values in the CSV file:
         date: 0
         bedrooms: 0
         bathrooms: 0
         sqft_living: 0
         sqft_lot: 0
         floors: 0
         waterfront: 0
         view: 0
         condition: 0
         sqft_above: 0
         sqft_basement: 0
         yr_built: 0
         yr_renovated: 0
         SalesPrice: 0
         no missing value
```

This sums up the null values in each column, and we can see there are 0s across each column, meaning no missing values.

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

Generate descriptive statistics. Descriptive statistics include those that summarize the central tendency, dispersion, and shape of a dataset's distribution, excluding NaN values.

```
In [9]: df1.describe()
```

Out[9]:

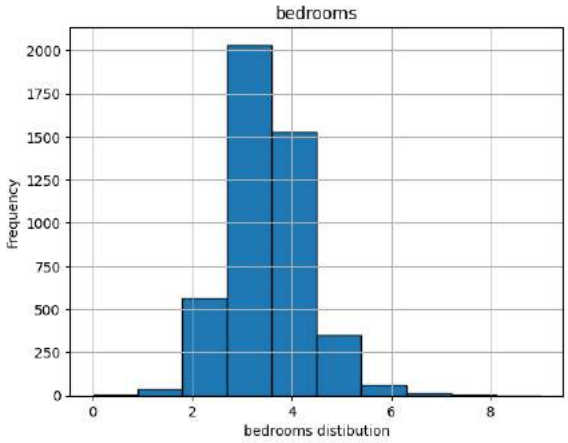
	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	sqft_above	sqft_basement	yr_built
count	4600.000000	4600.000000	4600.000000	4.600000e+03	4600.000000	4600.000000	4600.000000	4600.000000	4600.000000	4600.000000	4600.000000
mean	3.400870	2.160815	2139.346957	1.485252e+04	1.512065	0.007174	0.240652	3.451739	1827.265435	312.081522	1970.786304
std	0.908848	0.783781	963.206916	3.588444e+04	0.538288	0.084404	0.778405	0.677230	862.168977	464.137228	29.731848
min	0.000000	0.000000	370.000000	6.380000e+02	1.000000	0.000000	0.000000	1.000000	370.000000	0.000000	1900.000000
25%	3.000000	1.750000	1460.000000	5.000750e+03	1.000000	0.000000	0.000000	3.000000	1190.000000	0.000000	1951.000000
50%	3.000000	2.250000	1980.000000	7.683000e+03	1.500000	0.000000	0.000000	3.000000	1590.000000	0.000000	1976.000000
75%	4.000000	2.500000	2620.000000	1.100125e+04	2.000000	0.000000	0.000000	4.000000	2300.000000	610.000000	1997.000000
max	9.000000	8.000000	13540.000000	1.074218e+06	3.500000	1.000000	4.000000	5.000000	9410.000000	4820.000000	2014.000000

yr_renovated	SalesPrice
4600.000000	4.600000e+03
808.608261	5.519630e+05
979.414536	5.638347e+05
0.000000	0.000000e+00
0.000000	3.228750e+05
0.000000	4.609435e+05
1999.000000	6.549625e+05
2014.000000	2.659000e+07

5. Visualize the distribution of each feature

Bedrooms

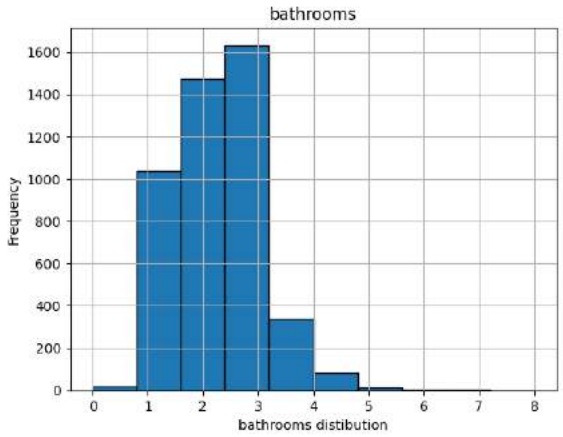
Bedrooms feature describes number of bedrooms a particular listed property has.



Number of bedrooms varies from 0 to 9. That is minimum a listed property can have 0 bedroom. And maximum a listed property can have 9 bedrooms. Most houses built have 3-4 bedrooms.

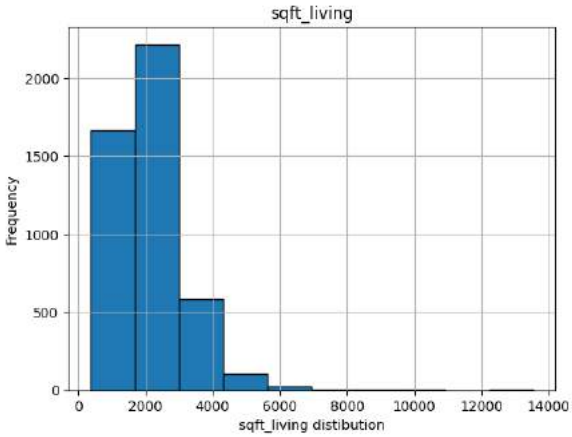
Bathrooms

Bathrooms features describe number of bathrooms a particular listed property have. Number of bathrooms varies from 0 to 8. That is a minimum a listed property can have 0 bathroom and maximum a listed property can have 8 bathrooms. Most houses built have between 2-3 bathrooms.



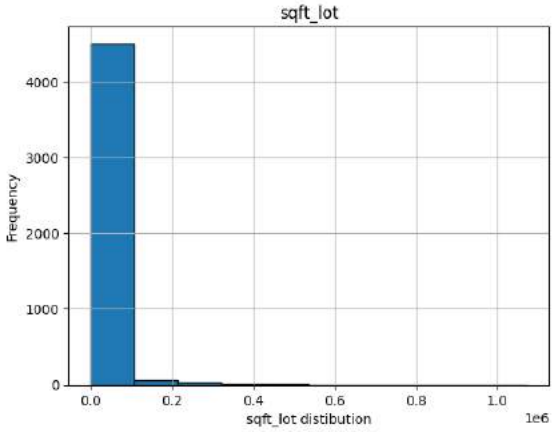
sqft_living

sqft_living features describe carpet area of living room of house. The built houses have sqft_living between 370 to 13540 sqft.



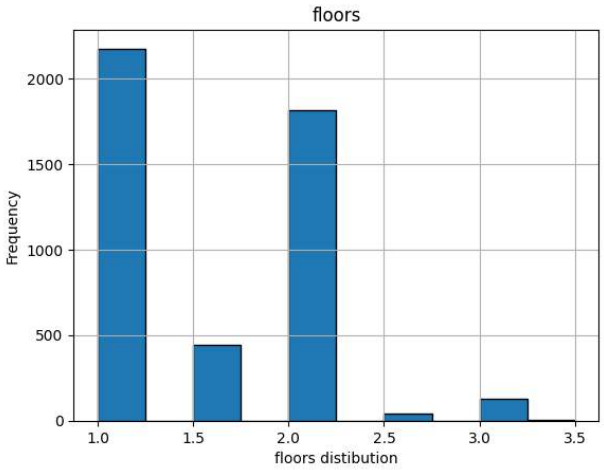
sqft_lot

sqft_lot features describe lot size area of house. The built houses have sqft_lot size between 638 to 1074218 sqft.



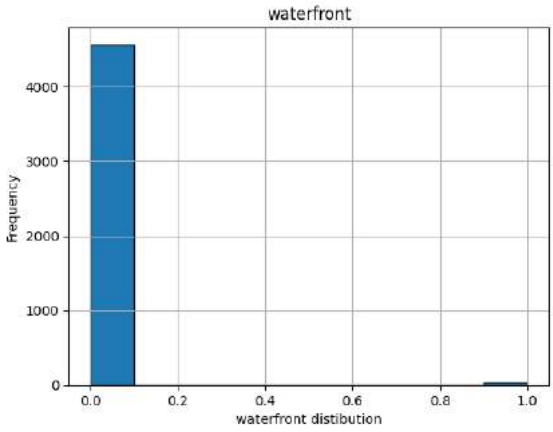
Floors

Floor feature describes number of floors a given house has. Number of floors varies from 1 to 3.5 i.e. a given house can have minimum 1 floor and maximum up to 3.5 floors. Most houses built have between 1 floor.



Waterfront

Binary indicator of whether the home has a view or direct access to a waterfront. 0 = No, 1 = Yes. From below distribution we can clearly see that maximum number of houses have no waterfront and there are very few which have waterfront.

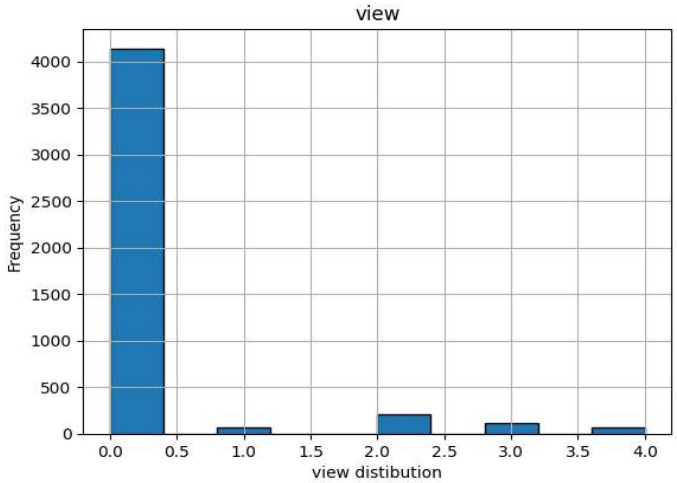


View

View indicates the quality of the view from the home. Some key details about the 'view' variable:

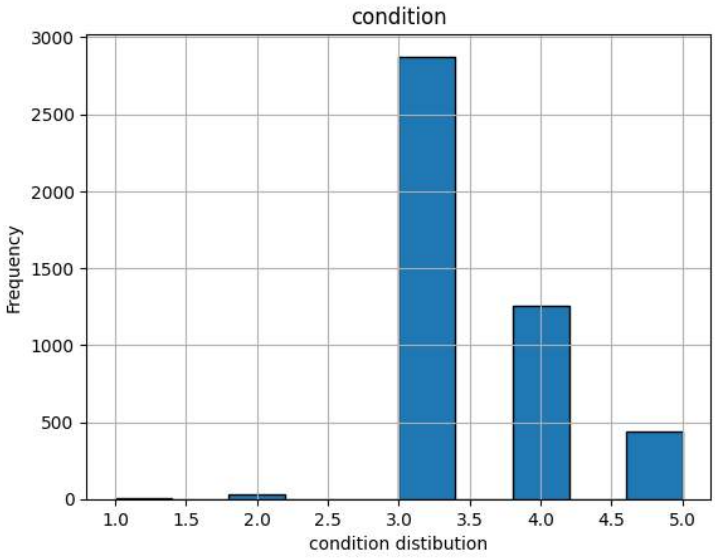
- It is an ordinal categorical variable that takes integer values from 0 to 4.
- 0 indicates no view.
- Higher values indicate better quality views, with 4 being the best possible view.
- This is a subjective rating of the aesthetic appeal of the view.

And from our frequency distribution we can clearly see that majority of houses have no view.



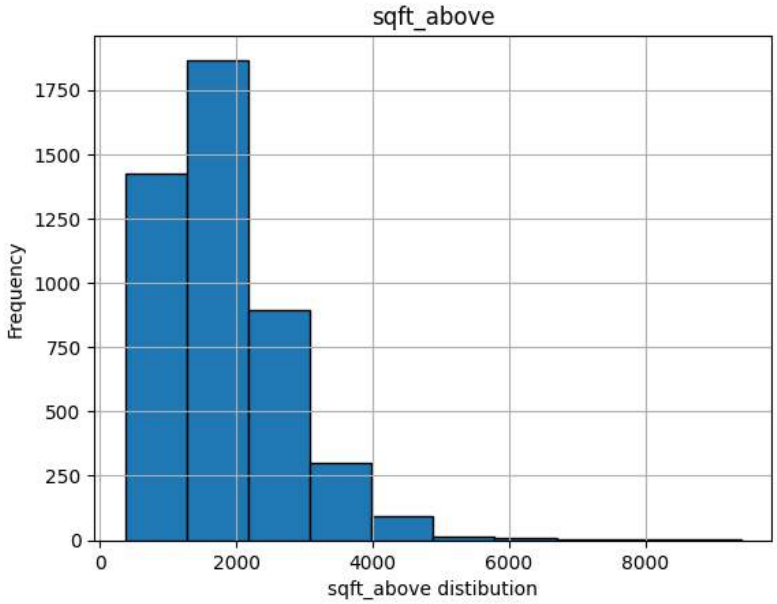
Condition

The condition in your dataset represents the condition of the property listed. A lower condition value may indicate that the property is in poor condition, while a higher condition value suggests that the property is in better condition. Majority of houses that are listed in market have condition 3.



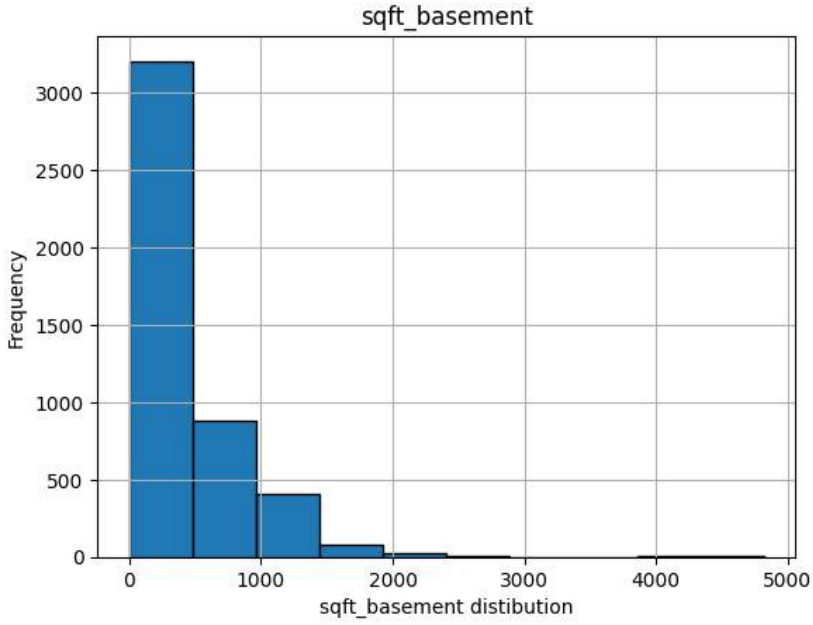
sqft_above

sqft_above measures the total livable floor area on the levels above ground level. The built houses have sqft_above size between 370 to 9410 sqft.



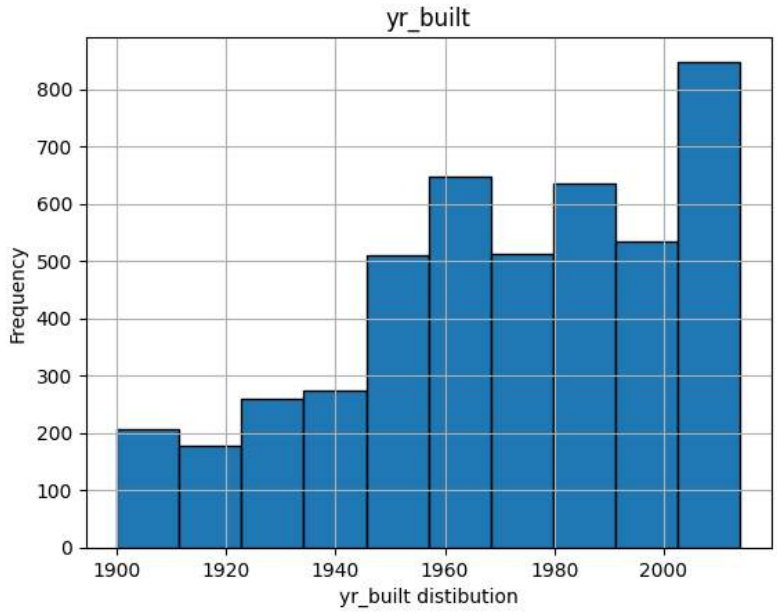
sqft_basement

It measures total floor area for basement. The built houses have sqft_basement size between 0 to 4820 sqft.



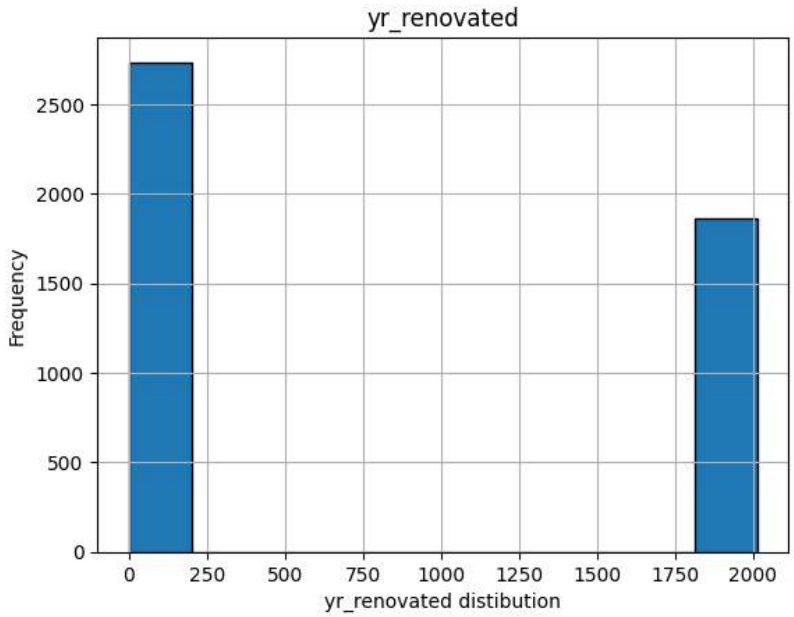
yr_built

Year in which listed property is built. Year in which house is built varies from 1900 to 2014.



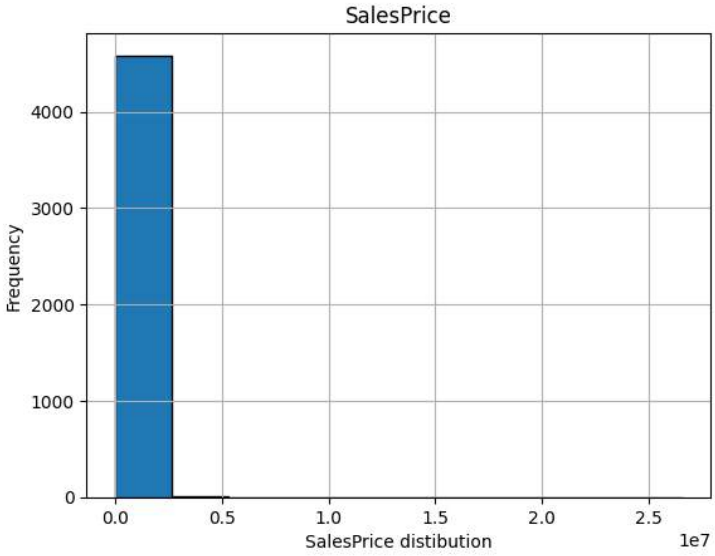
yr_renovated

year in which if listed property is renovated. Some values may be 0, indicating no renovations were performed. From our distribution we can clearly see that majority of houses have no renovations done to it.

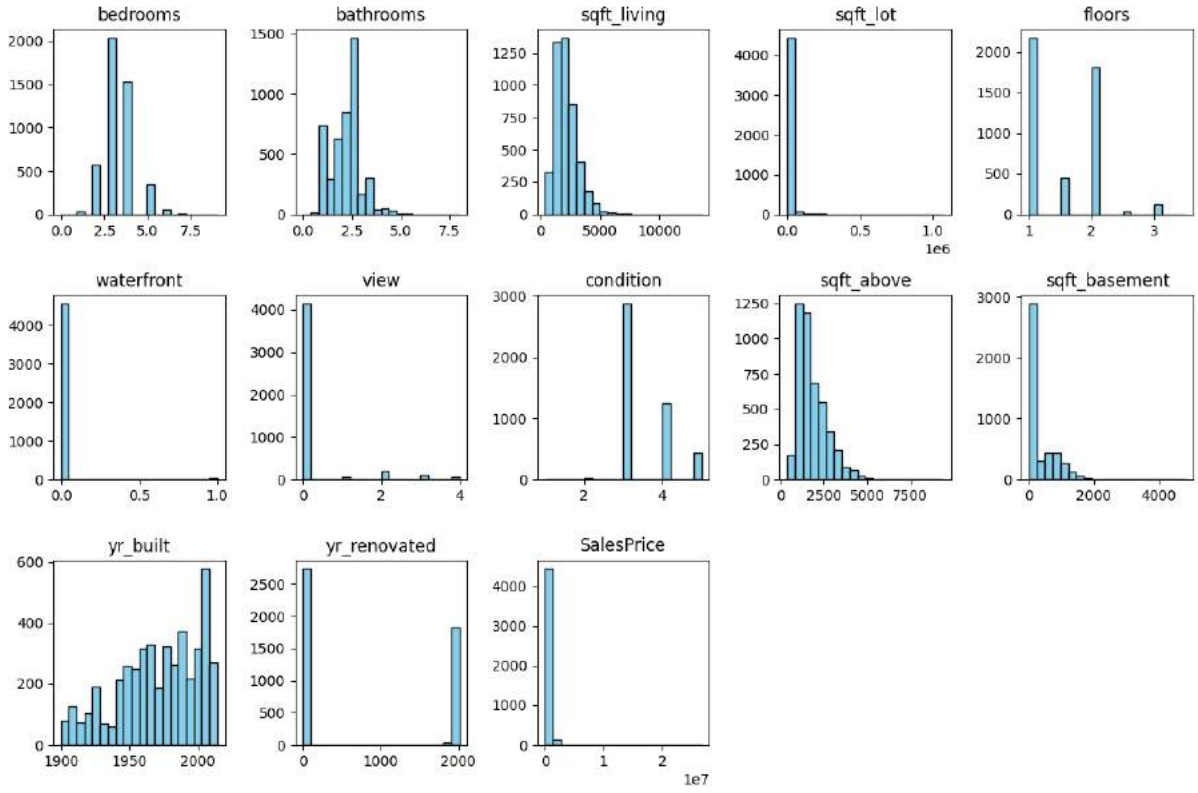


SalesPrice

Price of houses. Some properties have a price of 0, indicating missing or unknown values or severely distressed conditions. These properties could be government-owned, part of a land trust, or considered severely distressed.



Looking at every feature together



Linear Regression (Single Variable)

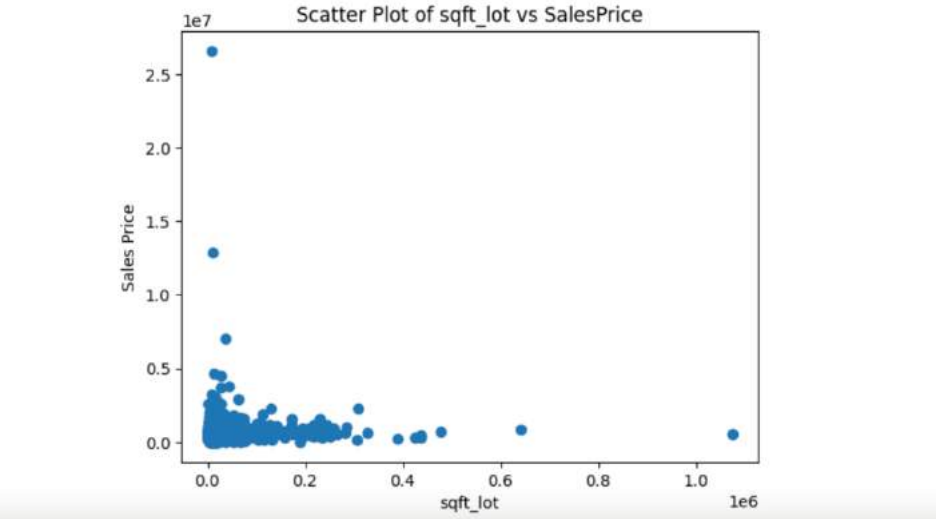
6. Implement your own linear regression model using the "sqft_lot" feature as the independent variable and "SalePrice" as the target variable. Print coef and intercept

Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable in this case

it is SalesPrice. The variable you are using to predict the other variable's value is called the independent variable in this case we are using sqft_lot as independent variable.

Distribution of **“sqft_lot”** with **“SalesPrice”** is shown below.

```
In [80]: x = np.array(df1[['sqft_lot']])

In [81]: y = np.array(df1[['SalesPrice']])

In [82]: plt.scatter(x,y)
         plt.xlabel("sqft_lot")
         plt.ylabel("Sales Price")
         plt.title("Scatter Plot of sqft_lot vs SalesPrice")
         plt.show()
```



The goal of making our own model to do linear regression is to find the line of best fit:

$$y = b0 + b1*x$$

Where: y = target variable

x = predictor variable

b0 = intercept

b1 = coefficient/slope

The coefficients b0 and b1 are estimated using the least squares method, which finds the line that minimizes the sum of squared residuals (difference between predicted and actual y).

The steps are:

1. Calculate the mean of x and y: x_mean, y_mean.
2. Calculate the numerator and denominator for the slope b1: Numerator = Σ(x - x_mean)*(y - y_mean) Denominator = Σ(x - x_mean)2 This sums over all samples.
3. Calculate b1 as numerator/denominator.
4. Calculate the intercept b0: b0 = y_mean - b1*x_mean

So, in summary, we are fitting a straight-line equation y = b0 + b1*x that minimizes the squared differences between the predicted and actual y values. The coefficients b0 and b1 are estimated from the data using the equations above.

```
In [40]: x_mean = np.mean(x)
         y_mean = np.mean(y)
         print("Mean for x values is", x_mean, "and mean for y values is", y_mean)

         Mean for x values is 14852.516086956522 and mean for y values is 551962.9884730434
```

to calculate value of coefficient and intercept

```
In [41]: num = 0
         den = 0

         for i in range(len(x)):
             num +=(x[i] - x_mean)*(y[i]-y_mean)
             den += (x[i]-x_mean)**2

         b1 = num/den
         b1

Out[41]: array([0.79271668])

In [42]: b0 = y_mean-(b1*x_mean)
         print("Intercept is ", float(b0), "Coefficient is", float(b1))

         Intercept is  540189.1512958274 Coefficient is 0.7927166756315349
```

Okay, with **coefficient of 0.7927** and **intercept of 540189.1512** here is an **interpretation**:

- The coefficient of 0.7927 indicates that for every 1 unit increase in sqft_lot, the model predicts a 0.7927 increase in SalesPrice. This is a stronger positive correlation.
- The intercept of 540189 indicates the expected SalesPrice when sqft_lot closes to 0, which suggests how high will be the price for small square feet space and the costly nature of the housing market.
- The larger coefficient indicates lot size has a stronger relationship with sales price in our model. Our model suggests a stronger positive correlation between lot size and price.

7. Calculate the sum of squared errors for your model

```
In [46]: # Calculate SSE
         sse = np.sum((y - y_pred)**2)
         print("Sum of Squared Errors (SSE) is", sse)

         Sum of Squared Errors (SSE) is 1458344675295682.5
```

Here are a few things we can interpret from Sum of Squared Errors (SSE) value of 1458344675295682.5:

- The SSE value is very large. This indicates the model is not fitting the data very well and has high error in its predictions.
- A good model will have a low SSE value close to 0. The lower the SSE, the better the fit.
- Some reasons for a high SSE could be:

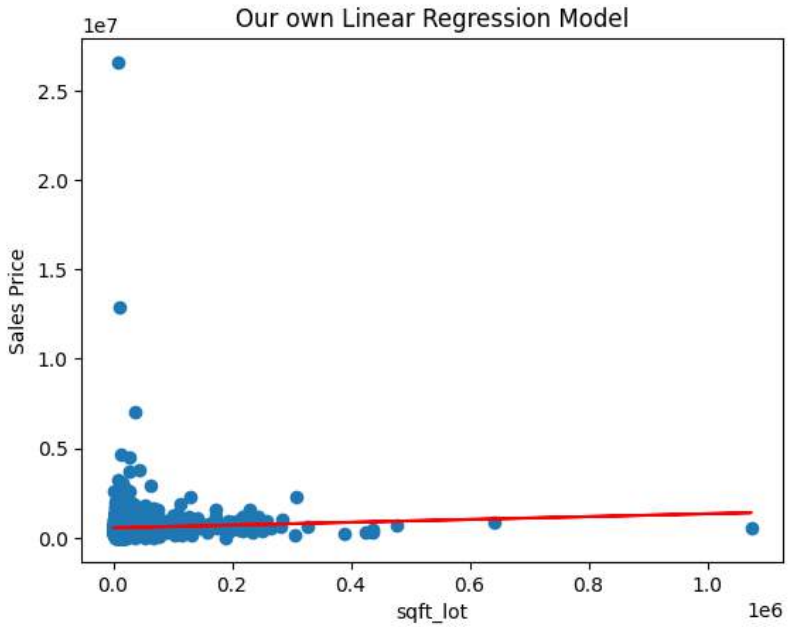
1. **Poor model selection** - A linear model may not fit data well. We could try higher degree polynomials, interaction terms, transformations etc.
2. **Overfitting** - The model may be fitting noise in the data too closely. Regularization could help.
3. **Outliers** - Extreme values can inflate the SSE. We could try removing/clipping outliers.
4. **Missing variables** - The model may be missing key predictor variables needed to accurately predict the target.

- To improve the model, we need to explore adding/removing predictors, using regularization, trying nonlinear models, or checking for outliers.

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

The below plot gives the actual data points (x, y) i.e., sqft_lot as x and SalesPrice as y as a scatter plot and overlays the regression line y_pred calculated using the coefficients b0 and b1. We can observe how well the regression line fits through the data points. For a good fit, the line should go through the centre of the data points.

The visualization helps to visually communicate the relationship between variables, assess the goodness of fit, and make predictions based on the regression model. Plotting the regression line alongside actual data points is a comprehensive approach that not only enhances the interpretability of statistical models but also facilitates effective communication of findings and insights derived from the analysis.



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

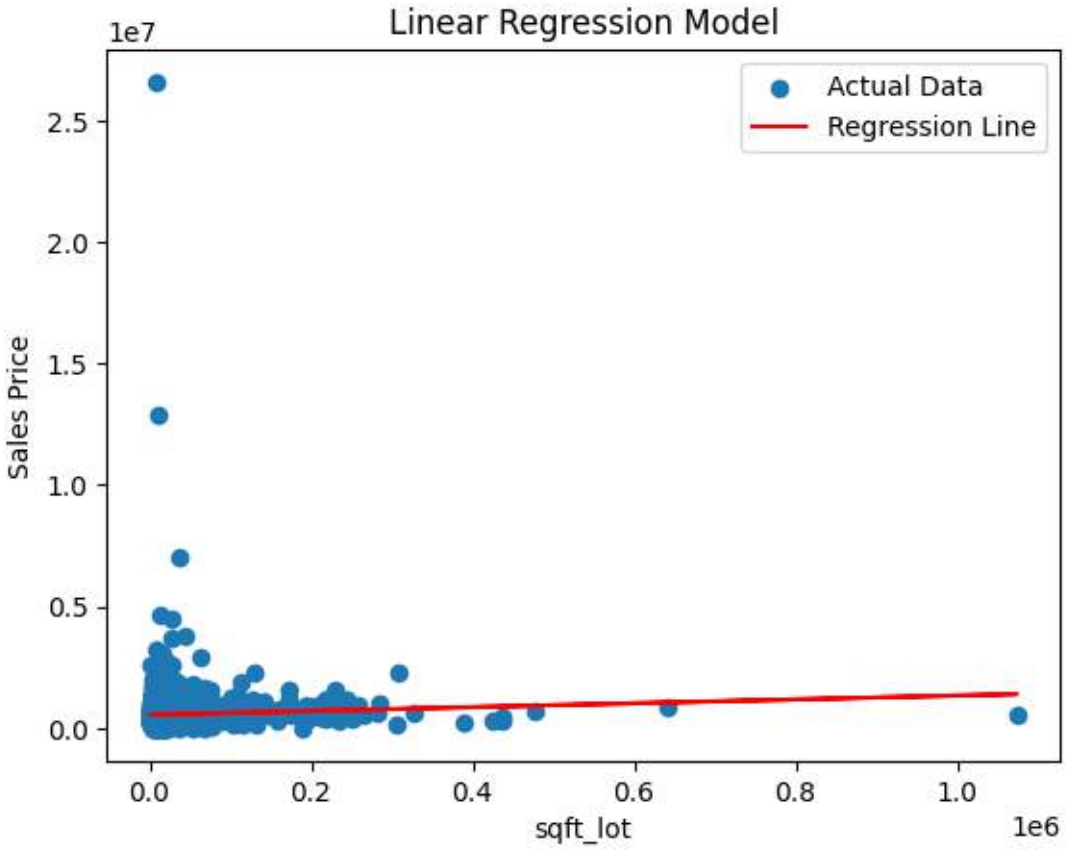
Scikit-learn (Sklearn) is Python's most useful and robust machine learning package. It offers a set of fast tools for machine learning and statistical modelling, such as classification, regression, clustering, and dimensionality reduction, via a Python interface.

Here are the coefficient and intercept of the regression line derived from the sklearn library.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Sklearn model
model = LinearRegression()
model.fit(x, y)
print("Sklearn coef:", model.coef_)
print("Sklearn intercept:", model.intercept_)
```

```
Sklearn coef: [[0.79271668]]
Sklearn intercept: [540189.15129583]
```

We can also visualize the predictions from the sklearn build model:



Comparison of results from both the libraries

```
# Compare coefficients and intercept
print("Manual Implementation:")
print("Intercept:", float(b0))
print("Coefficient:", float(b1))

print("\nSklearn Implementation:")
print("Intercept:", float(model.intercept_))
print("Coefficient:", float(model.coef_))
```

```
Manual Implementation:
Intercept: 540189.1512958274
Coefficient: 0.7927166756315349

Sklearn Implementation:
Intercept: 540189.1512958275
Coefficient: 0.7927166756315298
```

The small differences in the last decimal places of intercept and coefficient values are generally acceptable, and the code seems to be providing consistent results between the manual and scikit-learn implementations.

Linear Regression (Multivariate)

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

Multivariate Linear Regression is a statistical method used to model the relationship between multiple independent variables and a single dependent variable. Unlike simple linear regression, which involves only one predictor variable, multivariate linear regression accommodates several predictors, providing a more realistic representation of complex relationships in the data.

Multivariate Linear Regression was employed to investigate the relationship between two key property features, square footage of living space (sqft_living) and square footage of the lot (sqft_lot), and the corresponding sales prices (SalesPrice).

```
x = df1[['sqft_living', 'sqft_lot']]
y = df1['SalesPrice']
```

```
# Sklearn model
model = LinearRegression()
model.fit(x, y)
print("Sklearn coef:", model.coef_)
print("Sklearn intercept:", model.intercept_)
```

```
Sklearn coef: [257.13000008  -0.66039049]
Sklearn intercept: 11681.165815590997
```

The choice of using both square footage of living space and lot size as features recognizes the multidimensional nature of property valuation. It captures how changes in these features collectively contribute to variations in sales prices.

The coefficients obtained from the model provide insights into the impact of each feature on sales prices. A positive coefficient for a feature suggests a positive correlation, signifying that an increase in the respective feature is associated with an increase in the sales price.

A positive coefficient for **sqft_living** suggests a **positive correlation**, indicating that an increase in living space is associated with an increase in sales prices. The negative coefficient for **sqft_lot** suggests a **negative correlation**, implying that larger lot sizes may be associated with lower sales prices.

The intercept represents the predicted sales price when both sqft_living and sqft_lot are zero. In the context of real estate, this may not have practical significance but serves as a baseline reference point.

11. R-squared (R^2) score

Here is the R-squared score corresponding to the multivariable linear regression model.

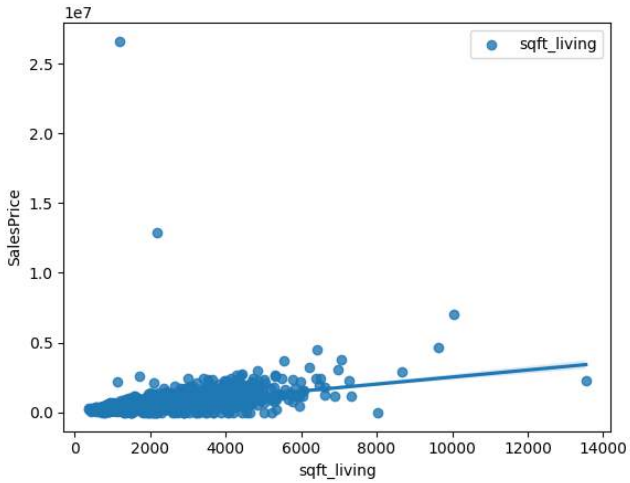
```
print('R2 score:', model.score(x, y))
```

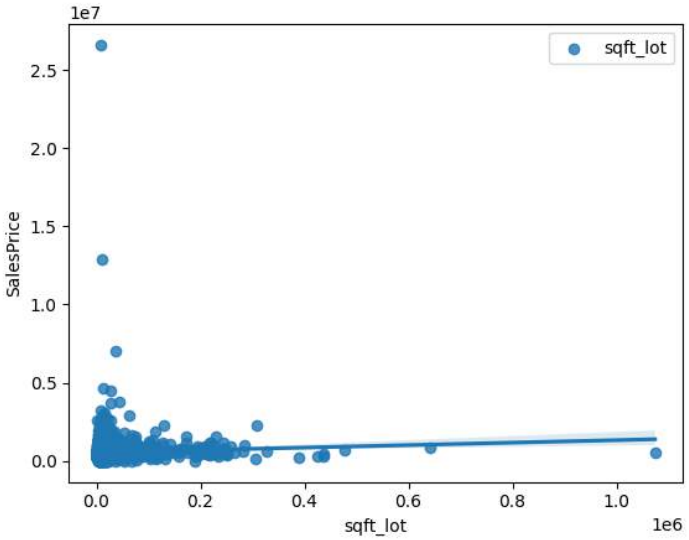
```
R2 score: 0.18694097425375722
```

R-squared information represents the proportion of the variance in the dependent variable that is predictable from the independent variables. This score indicates the proportion of variance in the sales prices that can be explained by our model.

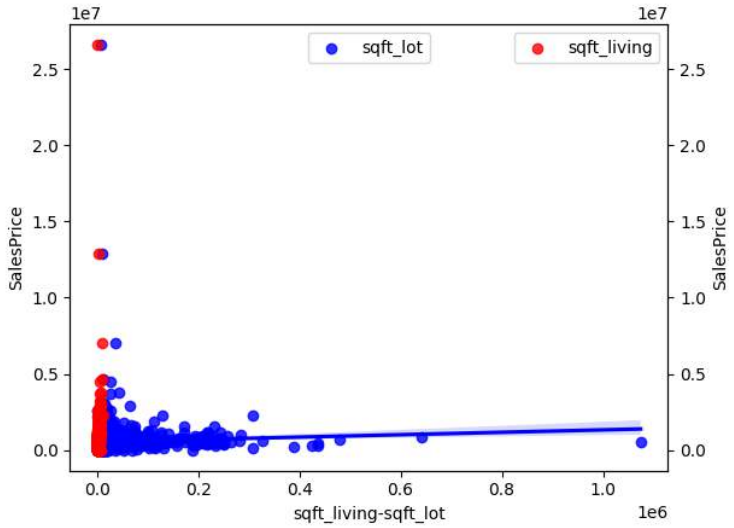
12. Relationships between the selected features and SalePrice

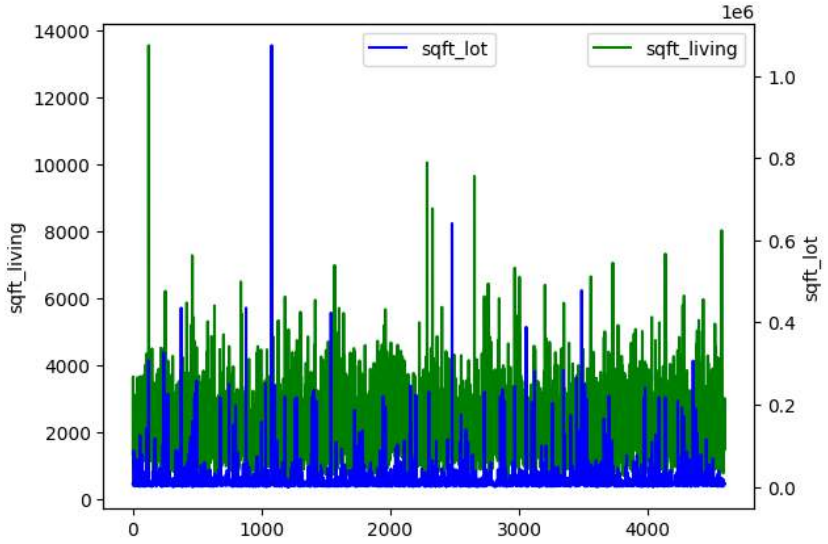
Feature dependence visualization independently



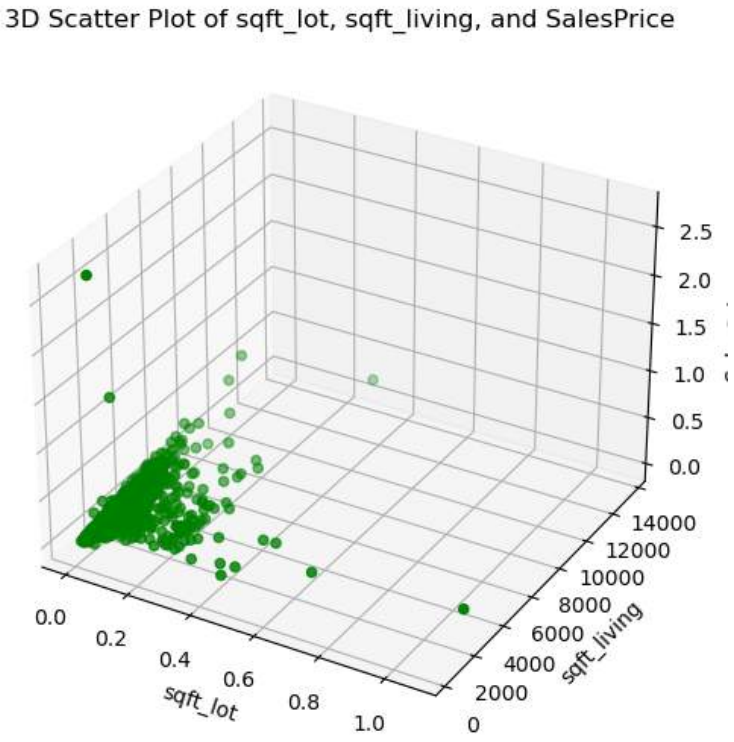


Feature dependence visualization together as combination chart (Dual axis):





Well, both above two graphs look busy. We can rather make a 3-D plot to see the data in a better manner.



Polynomial Regression

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

To further enhance the complexity of our regression analysis and accommodate potential non-linear patterns within our dataset, we employed a polynomial regression model. Specifically, we applied a polynomial feature transformation to the variable **sqft_lot**, introducing polynomial terms up to degree 2. This enables the model to better capture curvature or non-linear trends in the relationship between **sqft_lot** and the target variable.

This allows us to capture potential quadratic relationships between lot size and sales prices. This flexibility in modeling is particularly beneficial when the underlying patterns in the data exhibit curvature that cannot be adequately addressed by a simple linear model.

In real-world scenarios, property prices might not exhibit a strictly linear response to features like lot size. The implementation of a polynomial regression model allows us to refine our understanding of these relationships, providing a more nuanced perspective that can guide pricing strategies and investment decisions.

```
polynomial_features = PolynomialFeatures(degree = 2, include_bias = False)
Xp2 = polynomial_features.fit_transform(X)

pr2 = LinearRegression()
pr2.fit(Xp2,y)

#Coefficients

pr2.coef_

print("coefficients for degree 2 are : ", pr2.coef_)

#Intercept

pr2.intercept_

print("intercept for degree 2 is : ", pr2.intercept_)
```

```
coefficients for degree 2 are :  [[ 1.65511694e+00 -1.98125899e-06]]
intercept for degree 2 is :  [530368.09594368]
```

The coefficient for the linear term (**sqft_lot**) signifies the impact of lot size on sales prices. A positive coefficient suggests a positive correlation, meaning larger lot sizes are associated with higher sales prices. The coefficient for the squared term captures any non-linear influence, and its small negative value indicates a subtle concave relationship, suggesting that the impact of lot size may diminish for extremely large lots.

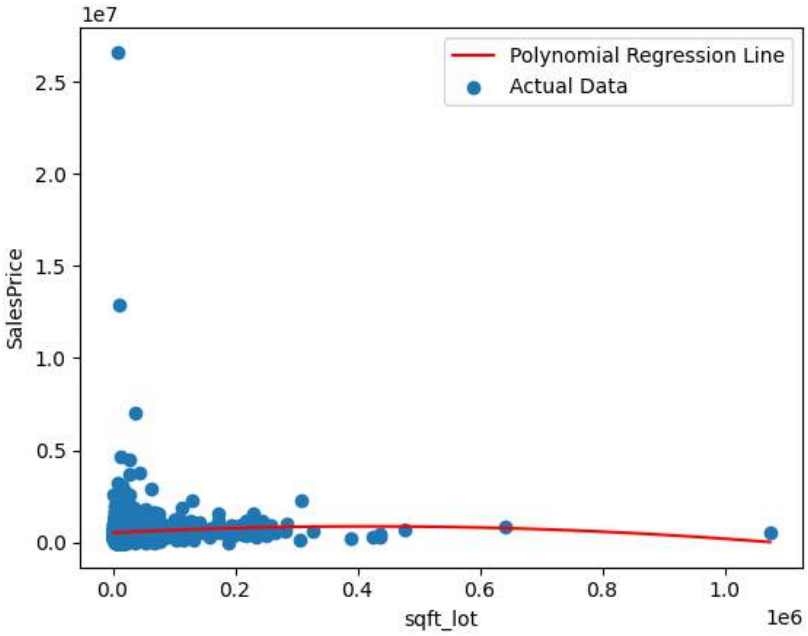
14. R-squared (R^2) score

The R-squared value for the polynomial regression model of degree 2 is 0.0045. This metric indicates that approximately 0.45% of the variability in sales prices can be explained by our model. While the R-squared value is modest, it provides context regarding the model's explanatory power in capturing the variance within the dataset.

```
R2_SLR = r2_score(y,pr2.predict(Xp2))
print("R2 for degree 2 : ",R2_SLR )
```

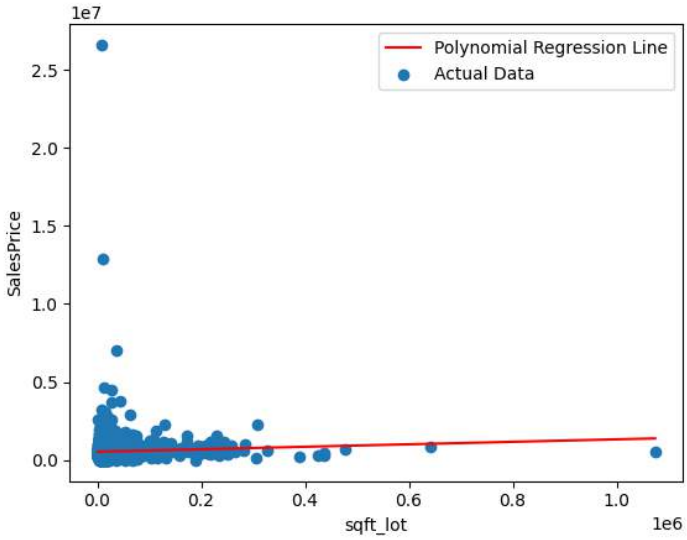
```
R2 for degree 2 :  0.00446670543314398
```

Relationship visualization for degree 2



15. Experiment with different polynomial degrees (Degree- 1,2,3,4,5,10,15)

Degree 1

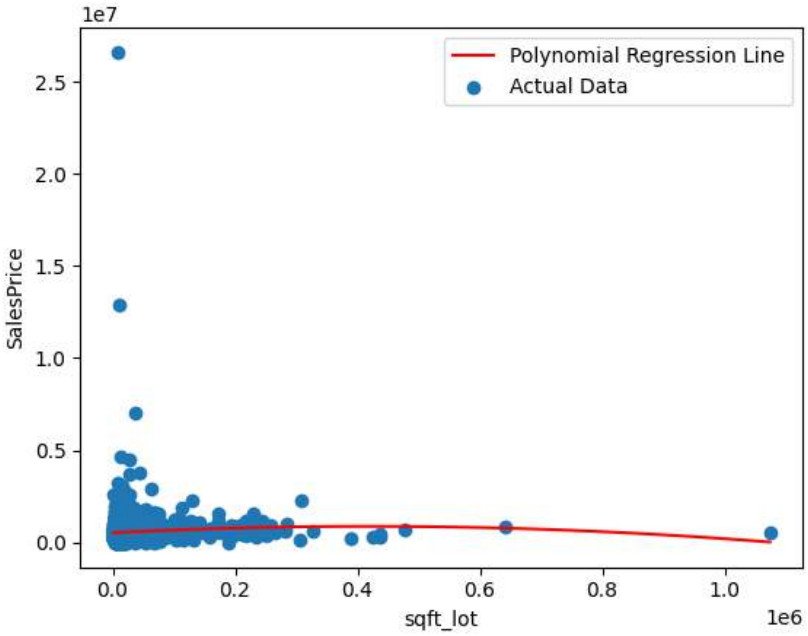



```
coefficients for degree 1 are :  [[0.79271668]]
intercept for degree 1 is :  [540189.15129583]
```

```
R2_SLR_degree1 = r2_score(y,pr1.predict(Xp1))
print("R2 for degree 1 : ",R2_SLR_degree1 )
```

```
R2 for degree 1 :  0.0025453331704339277
```

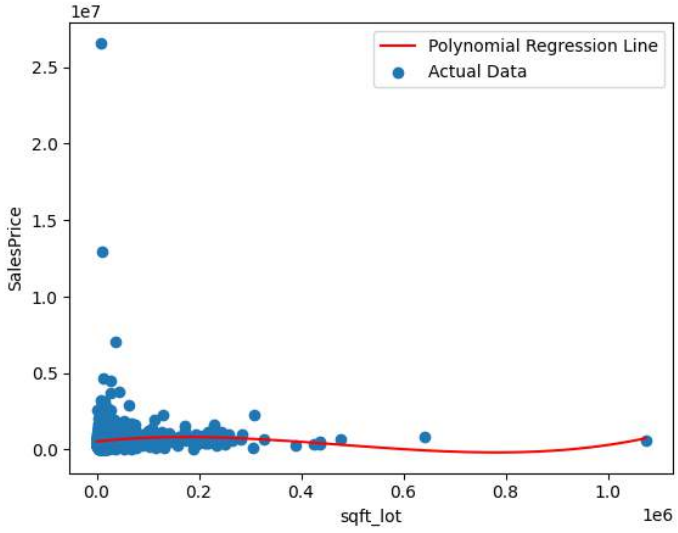
Degree 2




```
R2_SLR = r2_score(y,pr2.predict(Xp2))
print("R2 for degree 2 : ",R2_SLR )
```

```
R2 for degree 2 :  0.00446670543314398
```

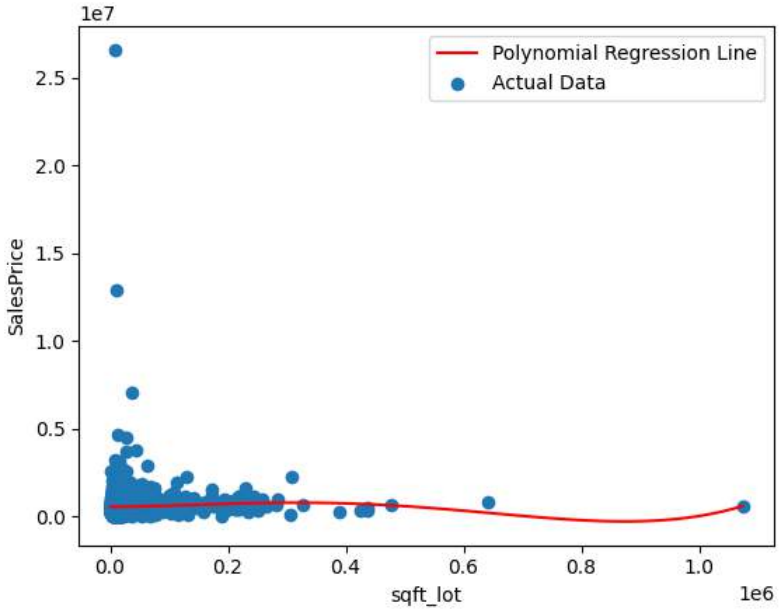
Degree 3




```
R2_SLR_degree3 = r2_score(y,pr3.predict(Xp3))
print("R2 for degree 3 : ",R2_SLR_degree3 )
```

```
R2 for degree 3 :  0.00710470276090347
```

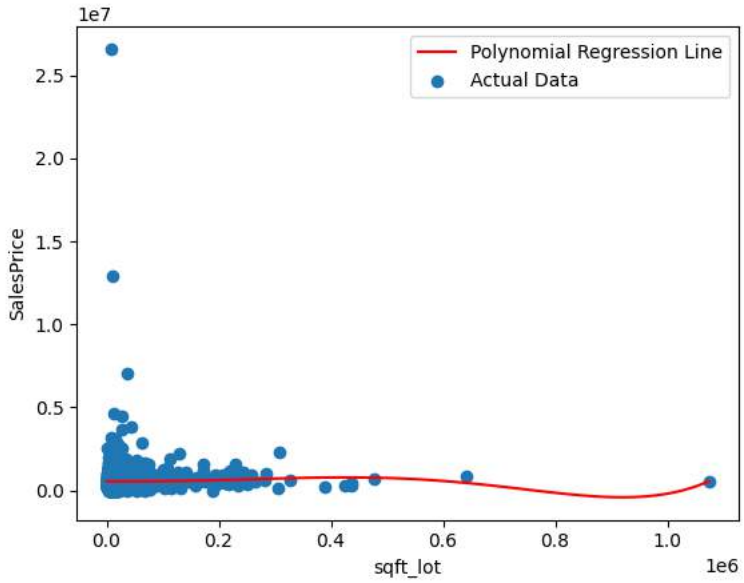
Degree 4




```
R2_SLR_degree4 = r2_score(y,pr4.predict(Xp4))
print("R2 for degree 4 : ",R2_SLR_degree4 )
```

```
R2 for degree 4 :  0.0017250387416192225
```

Degree 5

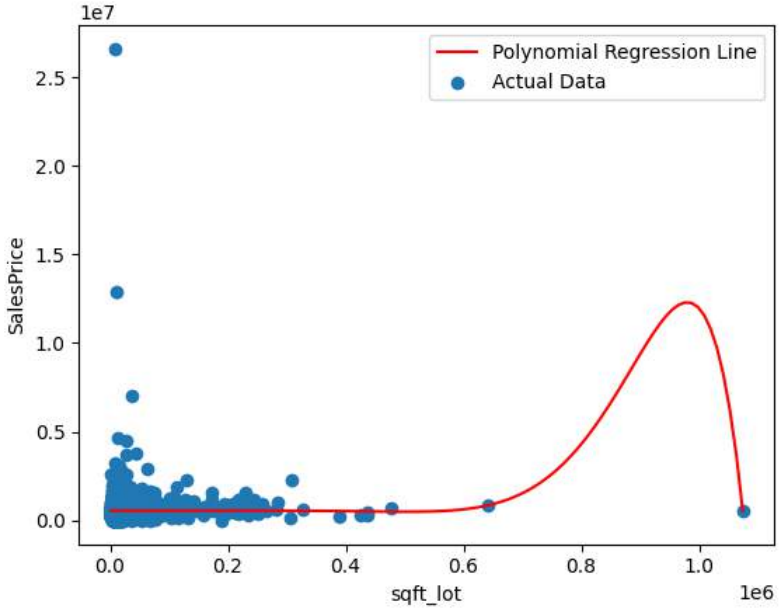


```
coefficients for degree 5 are :  [[ 4.91873832e-22  5.55705540e-15  1.68421738e-11 -4.33349944e-17
   2.57500342e-23]]
intercept for degree 5 is :  [550278.84087216]
```

```
R2_SLR_degree5 = r2_score(y,pr5.predict(Xp5))
print("R2 for degree 5 : ",R2_SLR_degree5 )
```

```
R2 for degree 5 :  0.0005669714676462823
```

Degree 10

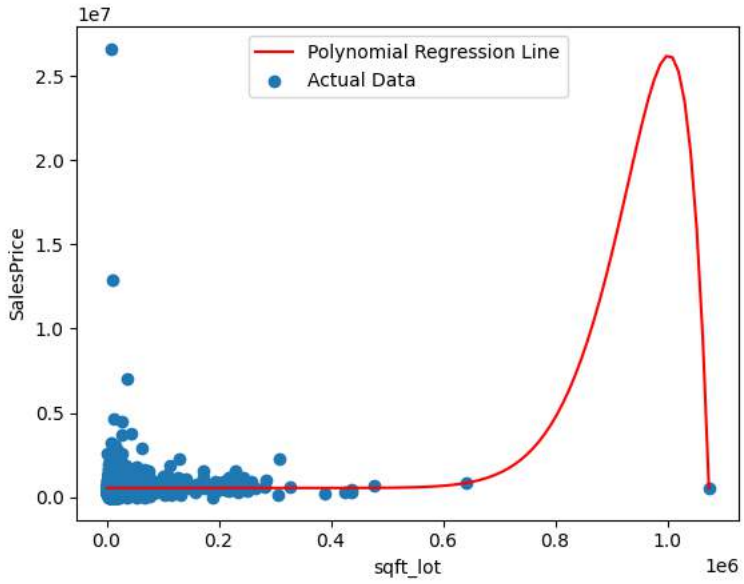



```
coefficients for degree 10 are :  [[-3.65498501e-79 -1.13201899e-41  1.21152311e-49  6.09940594e-76
  -9.49133300e-57 -3.43925683e-51 -1.05642764e-45 -2.23129742e-40
   5.96045538e-46 -3.61506246e-52]]
intercept for degree 10 is :  [551956.92356443]
```

```
R2_SLR_degree10 = r2_score(y,pr10.predict(Xp10))
print("R2 for degree 10 : ",R2_SLR_degree10 )
```

```
R2 for degree 10 :  6.630800154205918e-05
```

Degree 15



```
coefficients for degree 15 are :  [[ 2.98714332e-151 -4.84572979e-083 -2.82206518e-155  1.57667383e-133
   2.59885244e-113  6.85769384e-119  2.03938754e-116  1.29171522e-110
   8.12100603e-105  5.03781803e-099  3.05019899e-093  1.76325133e-087
   9.24177509e-082  3.71032154e-076 -3.45400657e-082]]
intercept for degree 15 is :  [551897.73894054]
```

```
R2_SLR_degree15 = r2_score(y,pr15.predict(Xp15))
print("R2 for degree 15 : ",R2_SLR_degree15 )
```

```
R2 for degree 15 :  6.041574340176492e-05
```

Find the best fit as per your perspective

Below R-squared values quantify the proportion of variance in sales prices that can be explained by our polynomial regression models. It's important to note that as the degree of the polynomial increases, we observe fluctuations in the model's ability to explain the variability in the target variable.

```
X = df1['sqft_lot'].values[:,np.newaxis]
y = df1['SalesPrice'].values

degrees = [1, 2, 3, 4, 5, 10, 15]

for degree in degrees:
    poly = PolynomialFeatures(degree=degree)
    X_poly = poly.fit_transform(X)

    poly_reg = LinearRegression()
    poly_reg.fit(X_poly, y)

    # Calculate R-squared
    r2 = r2_score(y, poly_reg.predict(X_poly))
    print("Degree {}: R-squared = {}".format(degree, r2))
```

```
Degree 1: R-squared = 0.0025453331704339277
Degree 2: R-squared = 0.00446670543314398
Degree 3: R-squared = 0.007104702760799664
Degree 4: R-squared = 0.0017250395682145703
Degree 5: R-squared = 0.0005669714665621495
Degree 10: R-squared = 6.6299993325191 76e-05
Degree 15: R-squared = 6.0415743401542876e-05
```

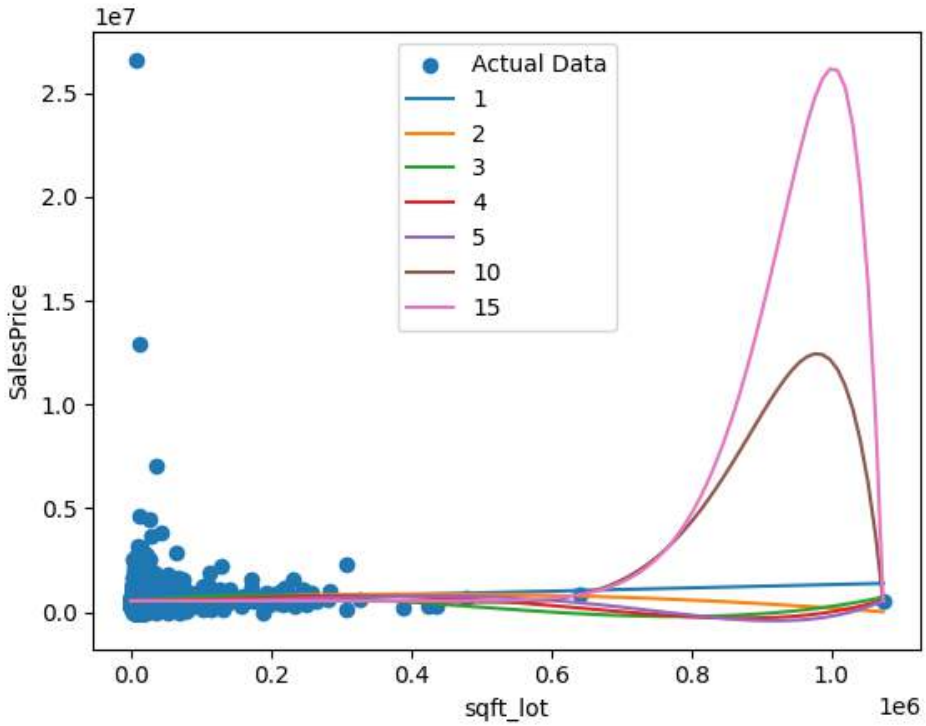
The R-squared values reveal the model's performance across different polynomial degrees. While a higher R-squared indicates a better fit, we must keep the trade off in mind. For our dataset, the explained variance remains relatively low across all polynomial degrees.

In our case, a polynomial of degree 3 might be a reasonable compromise between complexity and explanatory power.

16. Polynomial regression curve along with the actual data points

The combination of a scatter plot with a polynomial regression curve offers a comprehensive visual representation of the relationship between **sqft_lot** and sales prices. This approach aligns with our commitment to providing a holistic understanding of the dataset, leveraging both quantitative and visual insights to make informed decisions in the dynamic realm of real estate analysis.

Below is the graph showing plot for different degrees with actual data points:



The R-squared values are generally very low for all degrees, suggesting that the polynomial regression models are not capturing much of the variance in the data. However, among the given options,

Degree 3 has the highest R-squared value.

It's important to note that having a high degree polynomial doesn't necessarily mean a better fit. A model with too high a degree might be fitting the noise in the data rather than the underlying pattern, leading to poor generalization.

RANSAC (Robust Regression)

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

Employing RANSAC (Random Sample Consensus) to fit a robust linear regression model to the features **sqft_lot** and the target variable involves using a method that is less sensitive to outliers.

For resilient linear regression model for predicting sales prices based on **sqft_lot**, RANSAC (Random Sample Consensus) as a robust modeling technique is designed to handle datasets with potential outliers, a common challenge in real-world scenarios.

It addresses this challenge by iteratively fitting models to subsets of the data, identifying inliers and outliers, and ultimately selecting the model that best represents the consensus of most of the data.

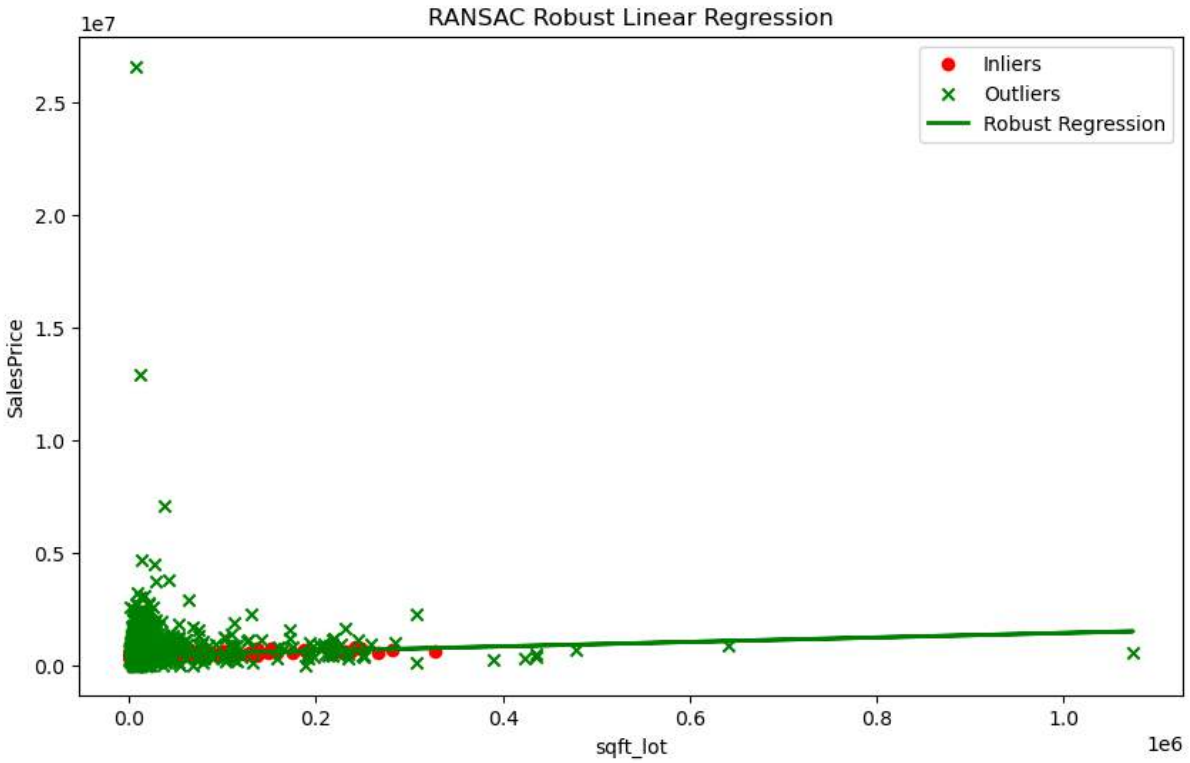
```
#Coefficients for ransac
print("Coefficients for ransac : ", ransac.estimator_.coef_)

#Intercept for ransac
print("Intercept for ransac : ",ransac.estimator_.intercept_)
```

```
Coefficients for ransac :  [0.94321897]
Intercept for ransac :  420801.72270645807
```

18. Co-ef and intercept. Visualize plot w.r.t. inliers and outliers.

```
#Coefficients for ransac
print("Coefficients for ransac : ", ransac.estimator_.coef_)

#Intercept for ransac
print("Intercept for ransac : ",ransac.estimator_.intercept_)
```

```
Coefficients for ransac :  [0.99257873]
Intercept for ransac :  425610.3352357076
```



These coefficient and intercept values reflect the consensus of most of the data, as determined by RANSAC, offering a resilient representation of the linear relationship between **sqft_lot** and sales prices.

19. R-squared (R^2) score with and without inliers

Full Data (With Inliers and Outliers Both)

```
# R-squared on full data (With Inliers)
R2_RANSAC = r2_score(y,ransac.predict(X))
print("R-squared on full data : ",R2_RANSAC )
```

```
R-squared on full data :  -0.040727236543824485
```

R-squared on full data (Without Inliers i.e. only outliers)

```
# Calculate R-squared for outliers only
outliers = ~inliers  # Invert the inliers mask to get outliers
y_pred_outliers = ransac.predict(X[outliers])
r2_outliers = r2_score(y[outliers], y_pred_outliers)
print(f'R-squared for RANSAC (outliers only): {r2_outliers:.4f}')
```

```
R-squared for RANSAC (outliers only): -0.0953
```

R-squared on full data (For Inliers only i.e. without Outliers)

```
# Calculate R-squared for inliers only
y_pred_inliers = ransac.predict(X[inliers])
r2_inliers = r2_score(y[inliers], y_pred_inliers)
print(f'R-squared for RANSAC (inliers only): {r2_inliers:.4f}')
```

```
R-squared for RANSAC (inliers only): 0.0636
```

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

Despite the generally low R-squared values across different polynomial degrees, it's notable that the **polynomial of degree 3** stands out as having the highest R-squared among the available polynomial models. While the R-squared values overall indicate that the models do not explain a significant portion of the variance in the data, the relatively higher R-squared for the degree 3 polynomial suggests that, among the polynomial options, it is performing comparatively better.

Part 3- Life Expectancy Prediction

1. Load the dataset and present the statistics of data

The dataset presents data about features which tells about the life expectancy based on provided features.

The dataset includes the following 22 features. These features encompass demographic, socioeconomic, and health-related variables, providing a comprehensive view of the factors that may impact life expectancy.

2. Reading Data into Pandas Dataframe

```
df1= pd.read_csv('LifeExpectancy.csv')
```

3. Initial Data Exploration

```
df1.head()
```

	Country	Year	Status	Life expectancy	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	...	Polio	Total expenditure	Diphtheria	HIV/AIDS	GD
0	Afghanistan	2015	Developing	65.0	263.0	62	0.01	71.279624	65.0	1154	...	6.0	8.16	65.0	0.1	584.2592
1	Afghanistan	2014	Developing	59.9	271.0	64	0.01	73.523582	62.0	492	...	58.0	8.18	62.0	0.1	612.6965
2	Afghanistan	2013	Developing	59.9	268.0	66	0.01	73.219243	64.0	430	...	62.0	8.13	64.0	0.1	631.7449
3	Afghanistan	2012	Developing	59.5	272.0	69	0.01	78.184215	67.0	2787	...	67.0	8.52	67.0	0.1	669.9590
4	Afghanistan	2011	Developing	59.2	275.0	71	0.01	7.097109	68.0	3013	...	68.0	7.87	68.0	0.1	63.5372

5 rows × 22 columns

Shape of the data and column names:

```
df1.shape
```

```
(2938, 22)
```

```
df1.columns
```

```
Index(['Country', 'Year', 'Status', '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')
```

Feature Information:

1. **Country:** Categorical variable representing the country.
2. **Year:** Numeric variable indicating the year of the recorded data.
3. **Status:** Categorical variable indicating the status of the country (e.g., developed or developing).
4. **Life expectancy:** Numeric variable representing the average expected lifespan.
5. **Adult Mortality:** Numeric variable indicating the mortality rate for adults.
6. **Infant deaths:** Numeric variable representing the number of infant deaths.
7. **Alcohol:** Numeric variable indicating the alcohol consumption per capita.

8. **Percentage expenditure:** Numeric variable indicating the percentage of total expenditure on health.
9. **Hepatitis B:** Numeric variable representing Hepatitis B immunization coverage among 1-year-olds.
10. **Measles:** Numeric variable representing the number of reported measles cases.
11. **BMI:** Numeric variable representing the Body Mass Index.
12. **Under-five deaths:** Numeric variable representing the number of deaths under the age of five.
13. **Polio:** Numeric variable representing Polio immunization coverage among 1-year-olds.
14. **Total expenditure:** Numeric variable representing the total health expenditure as a percentage of GDP.
15. **Diphtheria:** Numeric variable representing Diphtheria immunization coverage among 1-year-olds.
16. **HIV/AIDS:** Numeric variable representing the percentage of population with HIV/AIDS.
17. **GDP:** Numeric variable representing the Gross Domestic Product.
18. **Population:** Numeric variable representing the population of the country.
19. **Thinness 1-19 years:** Numeric variable representing the prevalence of thinness among individuals aged 1-19 years.
20. **Thinness 5-9 years:** Numeric variable representing the prevalence of thinness among individuals aged 5-9 years.
21. **Income composition of resources:** Numeric variable representing the income composition of resources.
22. **Schooling:** Numeric variable representing the average number of years of schooling.

2. Identify and specify the target variable from the dataset

The target variable is the column labeled **"Life expectancy".** The "Life expectancy" column represents the average expected lifespan in each entry of the dataset.

```
life_expectancy_description = df1['Life expectancy'].describe()

print(life_expectancy_description)
```

```
count    2938.000000
mean       69.234717
std         9.509115
min        36.300000
25%        63.200000
50%        72.100000
75%        75.600000
max        89.000000
Name: Life expectancy, dtype: float64
```

```
life_expectancy_head = df1['Life expectancy'].head(5)

print(life_expectancy_head)
```

```
0    65.0
1    59.9
2    59.9
3    59.5
4    59.2
Name: Life expectancy, dtype: float64
```

3. Categorize the columns into categorical and continuous

This categorization is essential for understanding the nature of your data and for making informed decisions during data analysis and modeling. Categorical variables are typically used for grouping and comparison, while numerical variables are used for quantitative analysis and modeling.

For year column, it can be both categorical or numerical depending on the intended use. In our context of time-related data like 'Year,' it's common to treat it as a numerical variable, especially if you plan to use it in time series analysis or regression models where the temporal ordering matters.

```
variable_types = df1.dtypes

# Categorize the variables based on their types
categorical_vars = variable_types[variable_types == 'object'].index.tolist()
numerical_vars = variable_types[variable_types != 'object'].index.tolist()

print("Categorical Variables:")
print(categorical_vars)

print("Numerical Variables:")
print(numerical_vars)
```

```
Categorical Variables:
['Country', 'Status']
Numerical Variables:
['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 year
s', 'thinness 5-9 years', 'Income composition of resources', 'Schooling']
```

4. Identify the unique values from each column

This information is valuable for understanding the diversity and distribution of data within each feature.

```
unique_counts = df1.nunique()

print("\nNumber of unique values in each column:")
print(unique_counts)
```

```
Number of unique values in each column:
Country                           193
Year                               16
Status                              2
Life expectancy                   362
Adult Mortality                   425
infant deaths                     209
Alcohol                          1077
percentage expenditure           2328
Hepatitis B                        87
Measles                           958
BMI                               608
under-five deaths                 252
Polio                              73
Total expenditure                 819
Diphtheria                         81
 HIV/AIDS                         200
GDP                              2491
Population                       2279
thinness  1-19 years              200
thinness 5-9 years                207
Income composition of resources   625
Schooling                         173
dtype: int64
```

5. 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

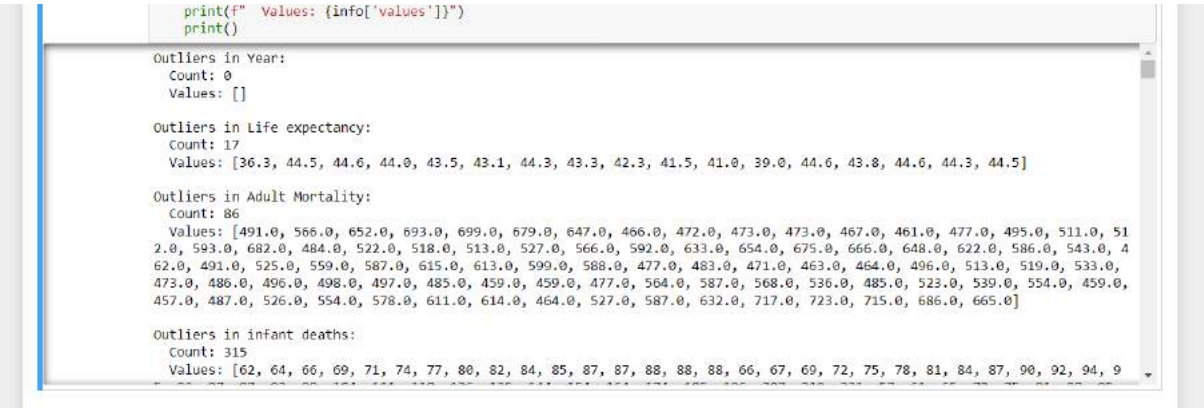
This information is important for ensuring the accuracy and reliability of your analysis, as missing values can introduce bias and impact the validity of your findings. Since we have a clean dataset without missing values, we can now proceed with your analysis with confidence.

```
Missing values in the CSV file:
Country: 0
Year: 0
Status: 0
Life expectancy: 0
Adult Mortality: 0
infant deaths: 0
Alcohol: 0
percentage expenditure: 0
Hepatitis B: 0
Measles: 0
BMI: 0
under-five deaths : 0
Polio: 0
Total expenditure: 0
Diphtheria: 0
 HIV/AIDS: 0
GDP: 0
Population: 0
thinness  1-19 years: 0
thinness 5-9 years: 0
Income composition of resources: 0
Schooling: 0
no missing value
```

The dataset under consideration contains no missing values. Each column, representing various features, has complete and available data for all 2938 entries. This absence of missing values eliminates the necessity for imputation techniques such as mean, median, or mode substitution.

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

The IQR method identifies outliers as data points that fall below the lower bound or exceed the upper bound.



```
    print(f"  Values: {info['values']}")
    print()

Outliers in Year:
  Count: 0
  Values: []

Outliers in Life expectancy:
  Count: 17
  Values: [36.3, 44.5, 44.6, 44.0, 43.5, 43.1, 44.3, 43.3, 42.3, 41.5, 41.0, 39.0, 44.6, 43.8, 44.6, 44.3, 44.5]

Outliers in Adult Mortality:
  Count: 86
  Values: [491.0, 566.0, 652.0, 693.0, 699.0, 679.0, 647.0, 466.0, 472.0, 473.0, 473.0, 467.0, 461.0, 477.0, 495.0, 511.0, 51
2.0, 593.0, 682.0, 484.0, 522.0, 518.0, 513.0, 527.0, 566.0, 592.0, 633.0, 654.0, 675.0, 666.0, 648.0, 622.0, 586.0, 543.0, 4
62.0, 491.0, 525.0, 559.0, 587.0, 615.0, 613.0, 599.0, 588.0, 477.0, 483.0, 471.0, 463.0, 464.0, 496.0, 513.0, 519.0, 533.0,
473.0, 486.0, 496.0, 498.0, 497.0, 485.0, 459.0, 459.0, 477.0, 564.0, 587.0, 568.0, 536.0, 485.0, 523.0, 539.0, 554.0, 459.0,
457.0, 487.0, 526.0, 554.0, 578.0, 611.0, 614.0, 464.0, 527.0, 587.0, 632.0, 717.0, 723.0, 715.0, 686.0, 665.0]

Outliers in infant deaths:
  Count: 315
  Values: [62, 64, 66, 69, 71, 74, 77, 80, 82, 84, 85, 87, 87, 88, 88, 88, 66, 67, 69, 72, 75, 78, 81, 84, 87, 90, 92, 94, 9
```

- Outliers in Year: Count: 0,
- Outliers in Life expectancy: Count: 17,
- Outliers in Adult Mortality: Count: 86,
- Outliers in infant deaths: Count: 315,
- Outliers in Alcohol: Count: 3,
- Outliers in percentage expenditure: Count: 389,
- Outliers in Hepatitis B: Count: 322,
- Outliers in Measles: Count: 542,

- Outliers in BMI: Count: 0,
- Outliers in under-five deaths : Count: 394,
- Outliers in Total expenditure: Count: 51 ,
- Outliers in Diphtheria: Count: 298,
- Outliers in HIV/AIDS: Count: 542,
- Outliers in GDP: Count: 445,
- Outliers in Population: Count: 452,
- Outliers in thinness 1-19 years: Count: 100,
- Outliers in thinness 5-9 years: Count: 99,
- Outliers in Income composition of resources: Count: 130,
- Outliers in Schooling: Count: 77

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

To address the presence of outliers in the dataset, a custom Python function was implemented to impute outlier values with the median for each numeric column.

Method:

a) The Interquartile Range (IQR) method was employed to identify outliers in each numeric column.
b) Outliers were replaced with the median of the respective column.
c) The provided function was applied iteratively to each numeric column in the dataset.

```
# Function to impute outliers with mean or median
def impute_outliers(column, impute_strategy='median'):
    q1 = column.quantile(0.25)
    q3 = column.quantile(0.75)
    iqr = q3 - q1

    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr

    # Identify outliers
    outliers = (column < lower_bound) | (column > upper_bound)

    # Impute outliers based on the specified strategy
    if impute_strategy == 'median':
        column[outliers] = column.median()
    elif impute_strategy == 'mean':
        column[outliers] = column.mean()
    else:
        raise ValueError("Invalid imputation strategy. Use 'median' or 'mean'.")

# Iterate through each column and impute outliers with mean or median
for column in df1.columns:
    if pd.api.types.is_numeric_dtype(df1[column]):
        # Decide imputation strategy based on column name
        if column in ['Alcohol', 'percentage expenditure', 'BMI', 'Total expenditure', 'Diphtheria', 'GDP', 'Population']:
            impute_strategy = 'mean'
        else:
            impute_strategy = 'median'

        impute_outliers(df1[column], impute_strategy)
```

To decide whether to impute with mean, median, or mode, you can consider the following guidelines:

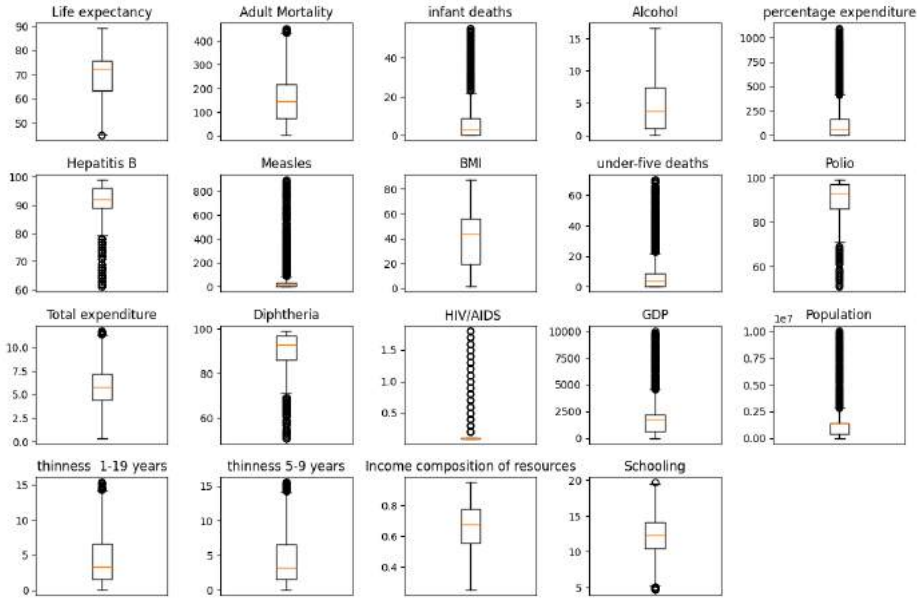
Mean: Use the mean for columns with a relatively symmetric distribution and no extreme outliers.

Examples from our data: Alcohol, percentage expenditure, BMI, Total expenditure, Diphtheria, GDP, Population.

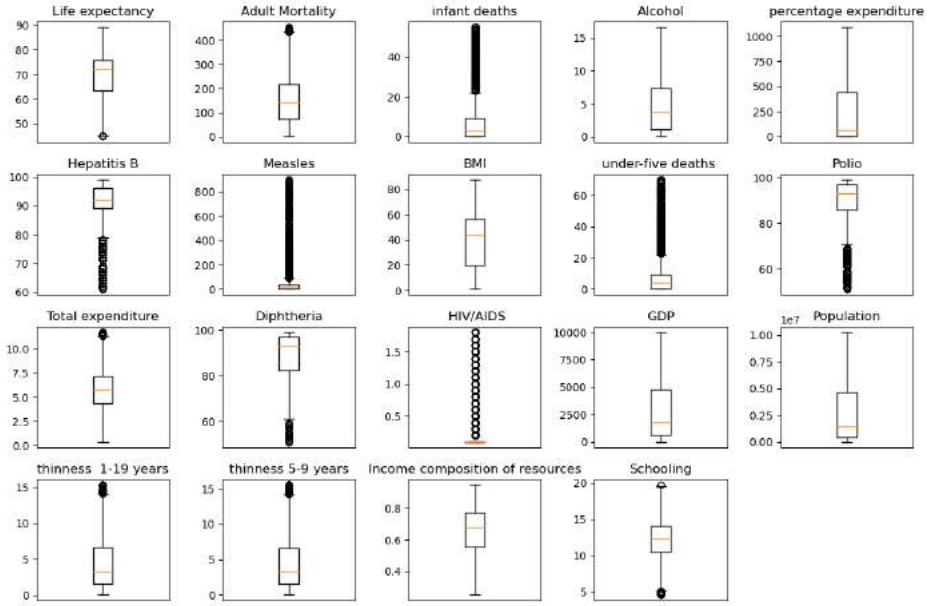
Median: Use the median for columns with a skewed distribution or in the presence of outliers.

Examples from our data: Adult Mortality, infant deaths, Measles, under-five deaths, HIV/AIDS, thinness 1-19 years, thinness 5-9 years.

Before imputation:



After imputation:

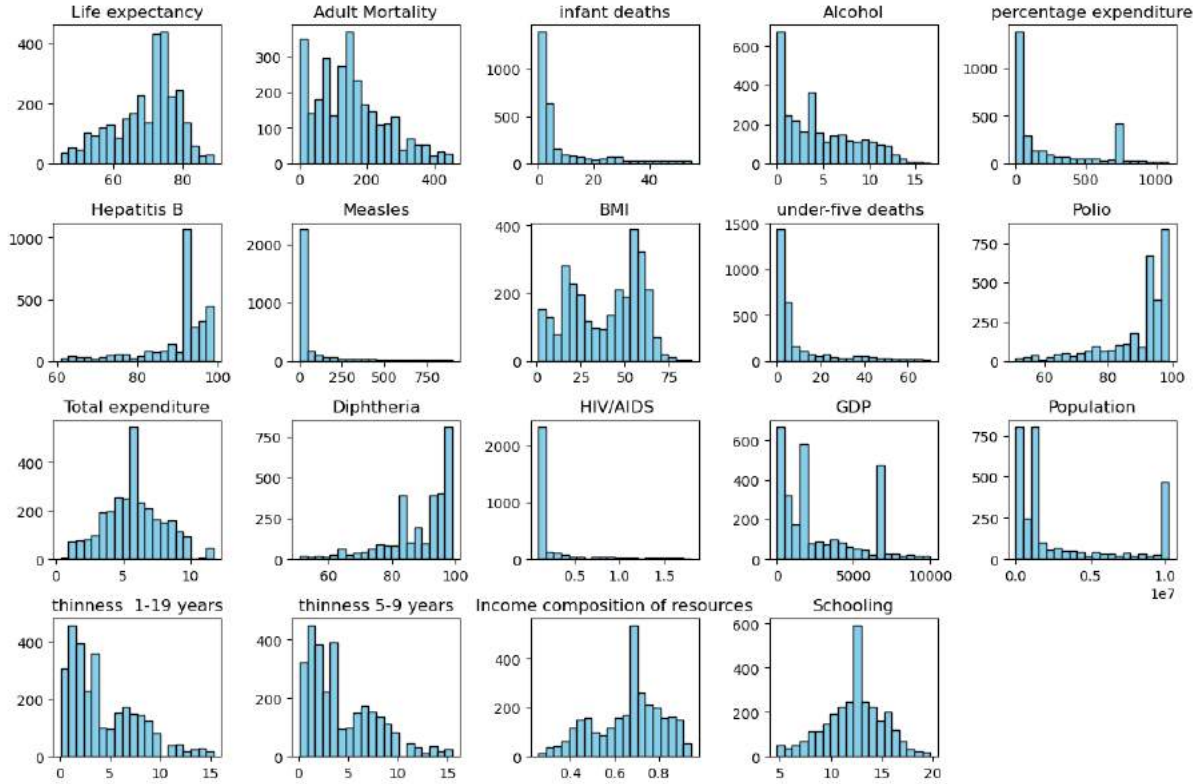


8. Summary statistics for numerical columns, such as mean, median, standard deviation

To gain a comprehensive understanding of the dataset, summary statistics were computed for the numerical columns. The statistics, including mean, median, and standard deviation, provide valuable insights into the central tendency and variability of each feature.

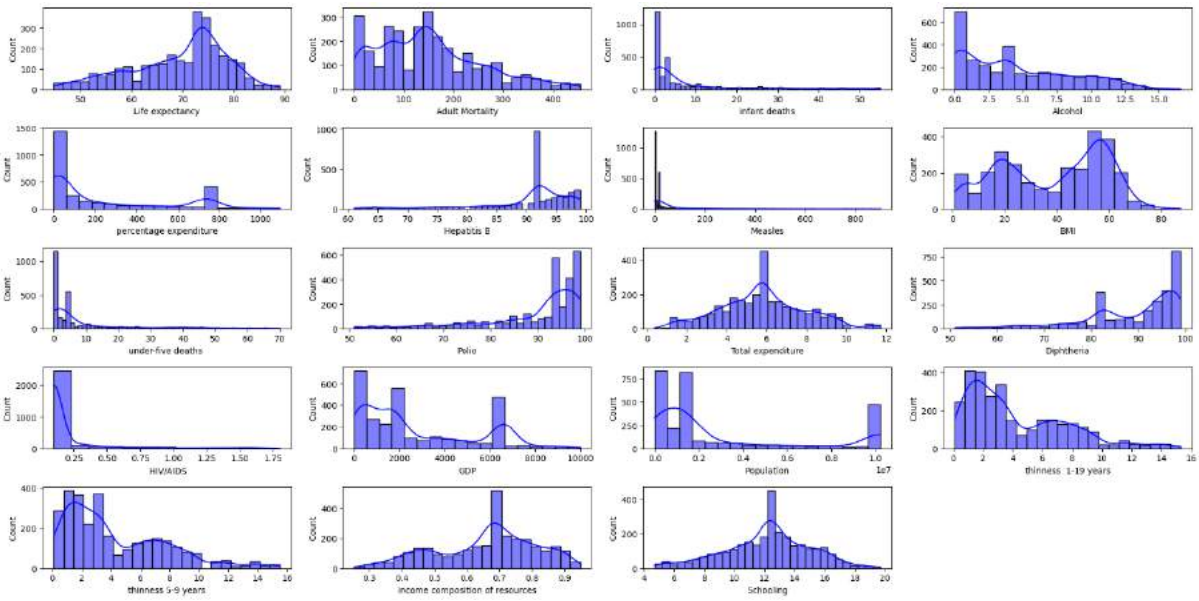
```
numerical_values_summary = df1[['Life expectancy', 'Adult Mortality', 'infant deaths', 'Alcohol', 'percentage expenditure',
                                'Hepatitis B', 'Measles', 'BMI', 'under-five deaths ', 'Polio', 'Total expenditure',
                                'Diphtheria', ' HIV/AIDS', 'GDP', 'Population', 'thinness  1-19 years',
                                'thinness 5-9 years', 'Income composition of resources', 'Schooling']].describe()
numerical_values_summary
```

	Life expectancy	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	BMI	under-five deaths	Polio	Total expenditure	Diph
count	2938.000000	2938.000000	2938.000000	2938.000000	2938.000000	2938.000000	2938.000000	2938.000000	2938.000000	2938.000000	2938.000000	2938.0
mean	69.403710	152.905990	8.059905	4.533761	236.571902	90.406399	70.735194	38.381178	9.115044	89.487406	5.788979	88.6
std	9.295013	103.551548	12.754371	3.900368	298.752003	8.278288	158.299914	19.935375	14.810333	10.783820	2.152299	10.4
min	44.800000	1.000000	0.000000	0.010000	0.000000	61.000000	0.000000	1.000000	0.000000	51.000000	0.370000	51.0
25%	63.425000	74.000000	0.000000	1.092500	4.685343	89.000000	0.000000	19.400000	0.000000	86.000000	4.370000	82.3
50%	72.100000	144.000000	3.000000	3.755000	64.912906	92.000000	17.000000	43.500000	4.000000	93.000000	5.755000	93.0
75%	75.600000	218.000000	9.000000	7.380000	441.534144	96.000000	36.000000	56.100000	9.000000	97.000000	7.150000	97.0
max	89.000000	454.000000	55.000000	16.580000	1092.155356	99.000000	899.000000	87.300000	70.000000	99.000000	11.710000	99.0



Features with a small standard deviation suggest less variability around the mean, while a larger standard deviation implies greater variability.

Comparing the mean and median can highlight potential skewness or asymmetry in the distribution of data.



'Life Expectancy' demonstrates a mean of 69.40, suggesting an overall positive outlook, yet with considerable variability (standard deviation of 9.30) indicative of differences among nations. The 'Adult Mortality' feature reflects a wide range of mortality rates (mean: 152.81, standard deviation: 103.55), underlining the diversity in healthcare and demographic patterns.

'Infant Deaths' present a mean of 8.06, with a substantial standard deviation of 12.75, emphasizing the varied healthcare conditions and disparities in child mortality. The 'Alcohol Consumption' feature displays moderate variability (standard deviation: 3.90) with a mean of 4.53, revealing differences in alcohol consumption patterns across populations. These findings are indicative of diverse cultural, social, and economic factors influencing health outcomes.

The features related to health infrastructure, such as 'Hepatitis B,' 'Measles,' 'Polio,' and 'Diphtheria,' exhibit mean values suggesting overall positive health practices. However, the standard deviations highlight disparities among countries, emphasizing the need for targeted interventions and healthcare improvements in specific regions.

Economic indicators like 'GDP' and 'Population' showcase wide-ranging values, with 'GDP' particularly displaying a broad spectrum of economic development. The 'Income Composition of Resources' and 'Schooling' features underscore the socio-economic dimensions influencing health outcomes, with mean values of 0.66 and 12.20.

9. Label encoding on certain columns

Specify and explain on which columns you perform and why.

Explain what label is encoding and how it changes the dataset.

Label encoding is a technique used **to convert categorical data** into numerical format, specifically integers. In this process, each unique category or label in a categorical variable is

assigned a unique integer. This conversion allows machine learning algorithms to work with categorical data, as most algorithms require numerical input.

- Assigning Integer Values: Each unique category in a categorical variable is mapped to a unique integer. The mapping is done in such a way that each category gets a distinct integer value. For example, if you have a categorical variable with three categories A, B, and C, label encoding might assign them the integers 0, 1, and 2, respectively.
- Ordinal Relationship: Label encoding assumes an ordinal relationship between the categories, meaning that the order of the integers reflects the order or ranking of the categories. This is crucial because some machine learning algorithms interpret the numerical values as having meaningful relationships.
- Application: Label encoding is often applied to categorical variables with ordinal characteristics, where the order among categories matters.

The columns 'Country' and 'Status' were chosen for this encoding process because Label encoding transforms categorical values into numerical representations, facilitating the incorporation of these features into predictive models.

The **LabelEncoder** from **the scikit-learn** library was employed to perform label encoding. This process assigns unique numerical labels to each distinct category within the specified columns.

```
#columns to be label encoded
columns_to_encode = ['Country', 'Status']

# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Apply label encoding to specified column
for column in columns_to_encode:
    if column in df1.columns:
        df1[column] = label_encoder.fit_transform(df1[column])

# Display the updated DataFrame
df1.head()
```

	Country	Year	Status	Life expectancy	Adult Mortality	infant deaths	Alcohol	percentage expenditure	Hepatitis B	Measles	...	Polio	Total expenditure	Diphtheria	HIV/AIDS	GDP	I
0	0	2015	1	65.0	263.0	3	0.01	71.279624	65.0	17	...	93.0	8.16	65.0	0.1	584.259210	1.0
1	0	2014	1	59.9	271.0	3	0.01	73.523582	62.0	492	...	58.0	8.18	62.0	0.1	612.696514	3.2
2	0	2013	1	59.9	268.0	3	0.01	73.219243	64.0	430	...	62.0	8.13	64.0	0.1	631.744976	1.0
3	0	2012	1	59.5	272.0	3	0.01	78.184215	67.0	17	...	67.0	8.52	67.0	0.1	669.959000	3.6
4	0	2011	1	59.2	275.0	3	0.01	7.097109	68.0	17	...	68.0	7.87	68.0	0.1	63.537231	2.9

5 rows × 22 columns

10. Data normalization on ‘Adult Mortality', 'BMI', 'GDP’ numerical columns using StandardScaler()

To ensure fair treatment of features with different scales, data normalization was applied to specific numerical columns. The features subjected to normalization include 'Adult Mortality,' 'BMI,' and 'GDP.' This process aims to scale the numerical values to a standardized range, mitigating the impact of differing scales on model performance.

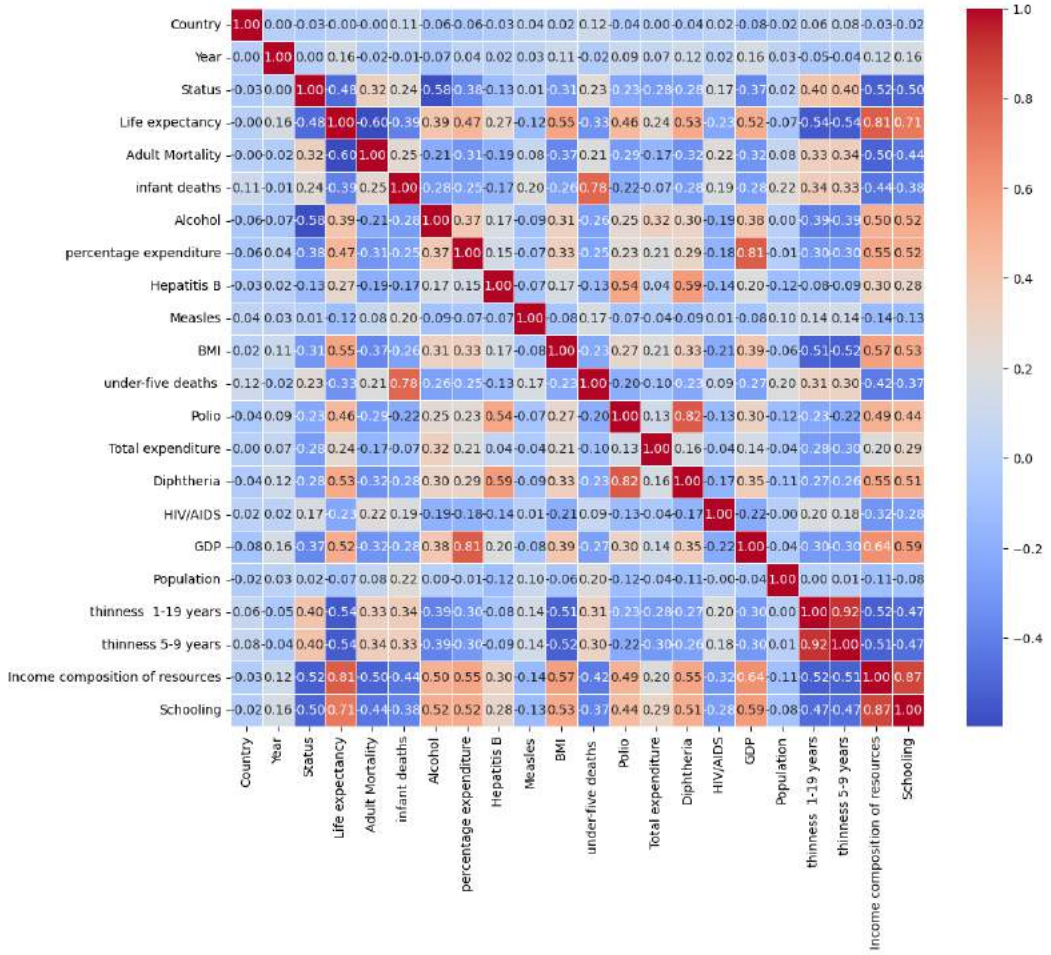
StandardScaler transforms the data by subtracting the mean and dividing by the standard deviation, resulting in a distribution with a mean of 0 and a standard deviation of 1. This normalization technique is particularly effective for features that exhibit varying magnitudes.

```
columns = ['Adult Mortality', 'BMI', 'GDP']

# Create StandardScaler instance
scaler = StandardScaler()

# Fit and transform selected columns
for col in columns:
    df1[col] = scaler.fit_transform(df1[[col]])

# View normalized columns
df1[columns].head()
```

	Adult Mortality	BMI	GDP
0	1.064328	-0.967349	-0.855836
1	1.141597	-0.992434	-0.844684
2	1.112621	-1.017519	-0.837214
3	1.151256	-1.042605	-0.822227
4	1.180232	-1.062673	-1.060050

11. Correlation matrix and plot the correlation using a heat map

To understand potential relationships and dependencies between different features in the dataset, a correlation matrix was computed. This matrix encapsulates the pairwise correlations between all numerical variables. Subsequently, these correlations were visually represented using a heatmap.



The matrix shows a numerical representation of the degree and direction of linear relationships between variables. Each entry in the matrix reflects the Pearson correlation coefficient, ranging from -1 (perfect negative correlation) to 1 (perfect positive correlation), with 0 indicating no linear correlation.

a) **The Features which are Most Positively Correlated with target variable.**

Year, country, Alcohol, percentage expenditure, Hepatits B, BMI, Polio, Total Expenditure, Diphtheria, GDP, Population, Income Composition of resources, Schooling

Year: The positive correlation with the 'Year' variable suggests a potential temporal influence on the target variable, indicating improvements or changes over time.

Country: 'Country' demonstrates a positive correlation, hinting at geographical variations impacting the target variable. This emphasizes the importance of considering country-specific factors in health outcomes.

Alcohol: The positive correlation with 'Alcohol' implies a potential link between alcohol consumption patterns and the target variable, underscoring the impact of lifestyle choices.

Percentage Expenditure: 'Percentage Expenditure' exhibiting a positive correlation suggests that increased healthcare expenditure as a percentage of GDP may contribute positively to the target variable.

Hepatitis B: A positive correlation with 'Hepatitis B' indicates a potential association between Hepatitis B vaccination coverage and the target variable, highlighting the importance of vaccination programs.

BMI (Body Mass Index): 'BMI' displaying a positive correlation suggests that variations in body mass may be linked to the target variable, emphasizing the role of nutrition and overall health.

Polio: The positive correlation with 'Polio' points to the significance of polio vaccination coverage in influencing the target variable, reflecting the impact of immunization efforts.

Total Expenditure: 'Total Expenditure' exhibiting a positive correlation suggests a potential relationship between total healthcare expenditure and the target variable, emphasizing the importance of healthcare investments.

Diphtheria: Positive correlation with 'Diphtheria' highlights the potential role of diphtheria vaccination coverage in contributing to the target variable, showcasing the impact of preventive measures.

GDP (Gross Domestic Product): 'GDP' showing a positive correlation suggests that economic prosperity, as reflected by GDP, may positively influence the target variable, reflecting broader socio-economic factors.

Income Composition of Resources: The positive correlation suggests that higher income composition positively impacts, emphasizing economic factors.

Schooling: 'Schooling' exhibiting a positive correlation underscores the role of education in positively influencing the target variable, emphasizing the importance of literacy and awareness.

b) The Features which are Most Negatively Correlated with target variable

Status, Adult Mortality, infant deaths, Measles, Under-five deaths, HIV/AIDS, thinness 1-19 years, thinness 5-9 years.

Status: The negative correlation with 'Status' indicates a potential impact of a country's development status on the target variable. Developing status might be associated with challenges that negatively influence health outcomes.

Adult Mortality: A strong negative correlation with 'Adult Mortality' underscores the critical role of adult mortality rates in shaping the target variable. Higher adult mortality rates may be indicative of challenges in healthcare and overall well-being.

Infant Deaths: 'Infant Deaths' displaying a negative correlation highlights the detrimental impact of high infant mortality rates on the target variable, emphasizing the vulnerability of early life stages.

Measles: Negative correlation with 'Measles' suggests that higher incidence of measles may negatively impact the target variable, emphasizing the importance of vaccination programs and disease prevention.

Under-five Deaths: The negative correlation with 'Under-five Deaths' reinforces the significance of reducing mortality rates among children under five years old to positively influence the target variable.

HIV/AIDS: The negative correlation with 'HIV/AIDS' underscores the adverse impact of higher HIV/AIDS prevalence on the target variable, highlighting the importance of effective prevention and treatment measures.

Population: Positive correlation with 'Population' indicates a potential relationship between the size of the population and the target variable, highlighting demographic influences.

Thinness 1-19 Years: Negative correlation with 'Thinness 1-19 Years' suggests that higher rates of thinness in the 1-19 age group may negatively influence the target variable, reflecting potential challenges in nutrition and health.

Thinness 5-9 Years: Similarly, negative correlation with 'Thinness 5-9 Years' emphasizes the adverse impact of thinness in the 5-9 age group on the target variable, signaling potential nutritional concerns.

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

The 'Country' column, identified as non-essential for predictive modeling in this context because of being categorical, was dropped from the dataset. This enhances the focus on relevant features that contribute more directly to the target variable, streamlining the dataset for more effective machine learning model training.

The dataset was divided into training and testing sets using a 70:30 split ratio. This ratio allocates 70% of the data for training the model, allowing it to learn patterns and relationships, while reserving 30% for testing, enabling evaluation on unseen data.

```
# Drop the 'Country' column
df = df1.drop('Country', axis=1)

# Life expectancy is the column we are trying to predict
X = df1.drop('Life expectancy', axis=1)
y = df1['Life expectancy']

# Split the dataset 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)

# Display the shapes of the training and testing sets
print("Training set shape:", X_train.shape, y_train.shape)
print("Testing set shape:", X_test.shape, y_test.shape)
```

```
Training set shape: (2056, 21) (2056,)
Testing set shape: (882, 21) (882,)
```

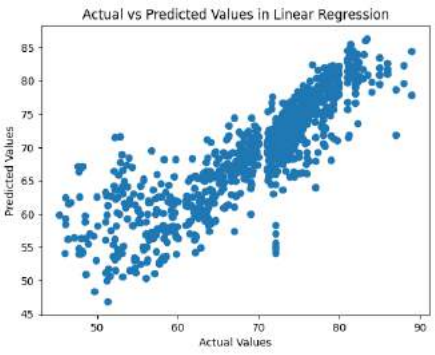
The 70:30 split ensures robust evaluation, allowing the model to be tested on data it has not seen during training, providing a more realistic assessment of its predictive performance.

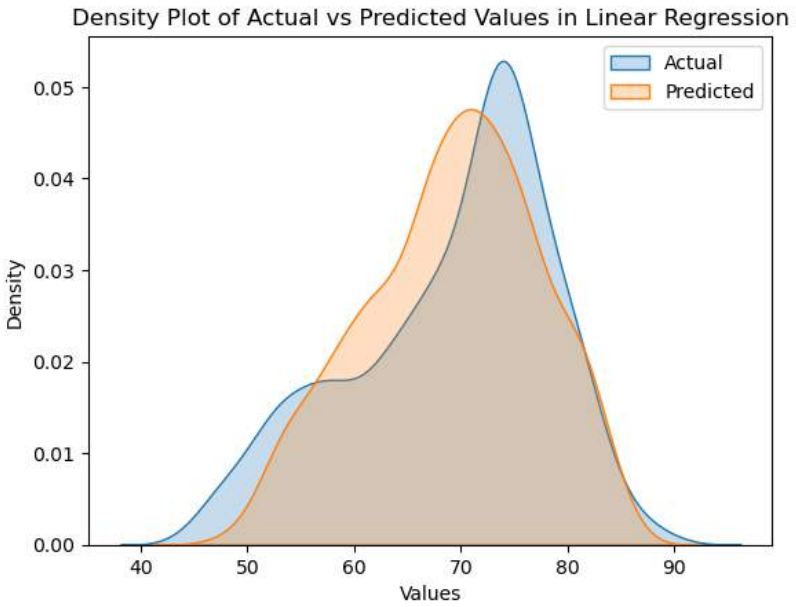
13. linear regression model using the training and testing datasets and compute mean absolute error

To predict 'Life Expectancy' based on the selected features, a linear regression model was constructed using the training dataset. This model leverages the relationships learned from the training data to make predictions on new, unseen instances. Subsequently, the model's performance was evaluated using the testing dataset, and the mean absolute error (MAE) was computed to gauge the accuracy of predictions.

```
# Create a 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 mean absolute error
mae = mean_absolute_error(y_test, y_pred)

# Display the mean absolute error
print("Mean Absolute Error:", mae)
```

```
Mean Absolute Error: 3.2798654240024003
```





The calculated MAE of 3.28 suggests that, on average, the model's predictions for 'Life Expectancy' deviated by approximately 3.28 units from the actual values. This provides a quantitative measure of the model's predictive performance.

Lower MAE values are indicative of higher prediction accuracy. In this case, a MAE of 3.28 implies that, on average, the model's predictions are within a reasonably close range of the true 'Life Expectancy' values.

14. 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

Two variants of gradient descent are employed: Mini-Batch Gradient Descent and Stochastic Gradient Descent. Both models are configured with specific hyperparameters, including alpha (regularization parameter), learning rate, maximum iterations, and batch size. The objective is to predict 'Life Expectancy' based on the chosen features, and the mean absolute error (MAE) was computed to assess the accuracy of the predictions.

Mini-Batch Gradient Descent is an optimization technique that strikes a balance between the efficiency of Batch Gradient Descent and the stochastic nature of Stochastic Gradient Descent. In this instance, the model was configured with an alpha of 0.0001, learning rate set to 'invscaling,' a maximum of 1000 iterations, and a batch size of 32.

```
# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Mini-batch gradient descent
alpha = 0.0001
learning_rate = 'invscaling'
max_iter = 1000
batch_size = 32

# Initialize the SGDRegressor for mini-batch gradient descent
mini_batch_model = SGDRegressor(alpha=alpha, learning_rate=learning_rate, max_iter=max_iter, random_state=42)

# Mini-batch gradient descent loop
for epoch in range(max_iter):
    for i in range(0, len(X_train_scaled), batch_size):
        X_batch = X_train_scaled[i:i+batch_size]
        y_batch = y_train.iloc[i:i+batch_size]
        mini_batch_model.partial_fit(X_batch, y_batch)

# Predictions for mini-batch
mini_batch_predictions = mini_batch_model.predict(X_test_scaled)
mini_batch_mae = mean_absolute_error(y_test, mini_batch_predictions)

# Display result
print("Mini-Batch Gradient Descent MAE:", mini_batch_mae)
```

```
Mini-Batch Gradient Descent MAE: 4.093399345954063
```

Stochastic Gradient Descent involves updating the model parameters based on the gradient of the loss function computed for a single training instance. The hyperparameters for the Stochastic Gradient Descent model were set to alpha=0.0001, learning rate='invscaling,' a maximum of 1000 iterations, and a batch size equivalent to one (stochastic).

```
# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Stochastic gradient descent
alpha = 0.0001
learning_rate = 'invscaling'
max_iter = 1000

# Initialize the SGDRegressor for stochastic gradient descent
stochastic_model = SGDRegressor(alpha=alpha, learning_rate=learning_rate, max_iter=max_iter, random_state=42)

# Stochastic gradient descent
stochastic_model.fit(X_train_scaled, y_train)

# Predictions for stochastic
stochastic_predictions = stochastic_model.predict(X_test_scaled)
stochastic_mae = mean_absolute_error(y_test, stochastic_predictions)

# Display result
print("Stochastic Gradient Descent MAE:", stochastic_mae)
```

```
Stochastic Gradient Descent MAE: 4.09941962020213
```

Results:

The lower MAE for Stochastic Gradient Descent (4.09) indicates a higher level of accuracy in predicting 'Life Expectancy' along with Mini-Batch Gradient Descent model (4.09).

The choice between Mini-Batch and Stochastic Gradient Descent depends on many factors such as dataset size or computational resources. Stochastic Gradient Descent, with its smaller batch size of one, often converges faster but may exhibit more erratic behavior.

15. Manual implementation of linear regression model using mini batch gradient descent with learning rate = 0.0001, maximum iterations =1000 and batch size=32

The manual implementation involves iteratively updating the model parameters using subsets (mini batches) of the training dataset.

```
import numpy as np

# Function to calculate mean absolute error
def mean_absolute_error(y_true, y_pred):
    return np.mean(np.abs(y_true - y_pred))

# Function for Mini-Batch Gradient Descent with L2 regularization
def mini_batch_gradient_descent(X, y, alpha=0.0001, learning_rate='invscaling', max_iter=1000, batch_size=32, random_state=None,
    m, n = X.shape
    theta = np.zeros((n, 1))

    if random_state:
        np.random.seed(random_state)

    for iteration in range(max_iter):
        indices = np.random.permutation(m)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        for i in range(0, m, batch_size):
            X_mini_batch = X_shuffled[i:i + batch_size]
            y_mini_batch = y_shuffled[i:i + batch_size]

            y_pred = np.dot(X_mini_batch, theta)
            gradient = -2 * np.dot(X_mini_batch.T, (y_mini_batch - y_pred)) / batch_size

            # Add L2 regularization term
            if penalty == 'l2':
                gradient += 2 * alpha * theta

            theta -= alpha * gradient
```

```
# Add bias term to X_train_scaled and X_test_scaled
X_train_b = np.c_[np.ones((len(X_train_scaled), 1)), X_train_scaled]
X_test_b = np.c_[np.ones((len(X_test_scaled), 1)), X_test_scaled]

# Apply Mini-Batch Gradient Descent on the training data with specified parameters
theta_mini_batch = mini_batch_gradient_descent(X_train_b, y_train_np, alpha=0.0001, learning_rate='invscaling', max_iter=1000, ba

# Predict on the test data
y_pred_test_mini_batch = np.dot(X_test_b, theta_mini_batch)

# Calculate Mean Absolute Error for Mini-Batch Gradient Descent
mae_mini_batch = mean_absolute_error(y_test_np, y_pred_test_mini_batch)

print(f"Mean Absolute Error (Mini-Batch Gradient Descent): {mae_mini_batch}")
```

```
Iteration 0, Train Mean Absolute Error: 68.45476779954069
Iteration 100, Train Mean Absolute Error: 18.945560997929096
Iteration 200, Train Mean Absolute Error: 6.060564025450126
Iteration 300, Train Mean Absolute Error: 3.5558016162581203
Iteration 400, Train Mean Absolute Error: 3.2200100176034385
Iteration 500, Train Mean Absolute Error: 3.157642999765713
Iteration 600, Train Mean Absolute Error: 3.1337352099644793
Iteration 700, Train Mean Absolute Error: 3.119802789431582
Iteration 800, Train Mean Absolute Error: 3.1103052356885557
Iteration 900, Train Mean Absolute Error: 3.1029913806841676
Mean Absolute Error (Mini-Batch Gradient Descent): 4.096485858791037
```

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

The results indicate that **standard linear regression** outperforms both mini-batch gradient descent and stochastic gradient descent in terms of MAE on your dataset.

Mini-batch and stochastic gradient descent are often used when dealing with large datasets, as they provide computational efficiency compared to the batch gradient descent. However, they may require more tuning of hyperparameters.

The choice between mini-batch and stochastic gradient descent depends on the dataset size, available computing resources, and the desired trade-off between computational efficiency and convergence stability.