Import the Dataset and Describe Summary Statistics

This step involves:

1. Importing the `house_prices.csv` dataset into a Pandas DataFrame.
2. Using the `.describe()` function to summarize the dataset's statistics.

In [2]:
```
import pandas as pd

# Import the dataset
data = pd.read_csv("C:/Users/suneh/Downloads/house_prices.csv")

#Show data
data.head()
```

Out[2]:

	Size (sq ft)	Number of Rooms	Neighborhood	Year Built	Price
0	3532	4	Suburb	1976	1195126.0
1	3407	5	Downtown	2010	1412375.0
2	2453	5	Countryside	1968	797476.0
3	1635	3	Downtown	1986	523051.0
4	1563	2	Suburb	1970	532291.0

In [4]:
```
# Display summary statistics
data.describe()
```

Out[4]:

	Size (sq ft)	Number of Rooms	Year Built	Price
count	500.000000	500.00000	500.000000	5.000000e+02
mean	2447.264000	3.05600	1986.268000	8.452926e+05
std	915.927716	1.39598	21.171695	3.295506e+05
min	807.000000	1.00000	1950.000000	7.771900e+04
25%	1672.750000	2.00000	1968.000000	5.945020e+05
50%	2449.000000	3.00000	1987.000000	8.447255e+05
75%	3237.750000	4.00000	2004.000000	1.105304e+06
max	3991.000000	5.00000	2022.000000	1.586530e+06

In [5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Size (sq ft)     500 non-null    int64  
 1   Number of Rooms  500 non-null    int64  
 2   Neighborhood     500 non-null    object 
 3   Year Built       500 non-null    int64  
 4   Price            500 non-null    float64
dtypes: float64(1), int64(3), object(1)
memory usage: 19.7+ KB
```

One-Hot Encoding for the Categorical Variable

The dataset contains a categorical variable `Neighborhood`. We'll apply one-hot encoding with `Number of Categories - 1` columns added for encoding. This ensures that we avoid multicollinearity.

In [15]:

```
# One-hot encoding for 'Neighborhood'
data_encoded = pd.get_dummies(data, columns=['Neighborhood'], drop_first=Tr

# Display the first few rows of the encoded DataFrame
data_encoded.head()
```

Out[15]:

	Size (sq ft)	Number of Rooms	Year Built	Price	Neighborhood_Downtown	Neighborhood_Sub
0	3532	4	1976	1195126.0	0	
1	3407	5	2010	1412375.0	1	
2	2453	5	1968	797476.0	0	
3	1635	3	1986	523051.0	1	
4	1563	2	1970	532291.0	0	

Correlation Matrix

We compute the correlation matrix to identify relationships between variables, focusing on strong correlations with `Price`.

In [20]:

```
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# Calculate the correlation matrix
correlation_matrix = data_encoded.corr()
```

In [22]:
```
correlation_matrix
```

Out[22]:

	Size (sq ft)	Number of Rooms	Year Built	Price	Neighborhoo
Size (sq ft)	1.000000	0.013477	-0.011671	0.908024	
Number of Rooms	0.013477	1.000000	-0.032377	0.231002	
Year Built	-0.011671	-0.032377	1.000000	0.169842	
Price	0.908024	0.231002	0.169842	1.000000	
Neighborhood_Downtown	0.002287	-0.084240	0.001649	0.003634	
Neighborhood_Suburb	0.018282	0.089848	-0.044649	0.021090	

The strongest indicator of house price is Size (sq ft) at 0.908024.

Create Age Variable and Remove Year Built

The age of the house in 2024 is calculated as `2024 - Year Built`. The `Year Built` column will then be removed.

In [39]:
```
# Calculate house age
data_encoded['Age'] = 2024 - data_encoded['Year Built']

# Drop 'Year Built'
data_encoded.drop(columns=['Year Built'], inplace=True)

# Display the first few rows of the modified DataFrame
data_encoded.head()
```

Out[39]:

	Size (sq ft)	Number of Rooms	Price	Neighborhood_Downtown	Neighborhood_Suburb	A
0	3532	4	1195126.0	0	1	
1	3407	5	1412375.0	1	0	
2	2453	5	797476.0	0	0	
3	1635	3	523051.0	1	0	
4	1563	2	532291.0	0	1	

Linear Regression Model

We train a linear regression model with the following steps:

1. Split the dataset into training and testing sets (80% train, 20% test, random state=18).
2. Train the model using the training set.
3. Display the model's intercept and coefficients.

In [45]:

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

# Define features and target variable
X = data_encoded.drop(columns=['Price'])
y = data_encoded['Price']
```

In [47]:

```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra

# Train the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)
```

Out[47]:

```
▾ LinearRegression
LinearRegression()
```

In [49]:

```
# Display model coefficients and intercept
print("Intercept:", model.intercept_)
print("Coefficients:", dict(zip(X.columns, model.coef_)))
```

```
Intercept: -16218.657343078754
Coefficients: {'Size (sq ft)': 327.3131214474899, 'Number of Rooms': 52678.0
1938654492, 'Neighborhood_Downtown': 15460.415653697008, 'Neighborhood_Subur
b': 10388.111962154166, 'Age': -2941.3611239281527}
```

Model Evaluation

We evaluate the model using Root Mean Squared Error (RMSE) on the test set.

In [55]:
```
# Predict on the test set
y_pred = model.predict(X_test)

# Calculate RMSE
rmse = mean_squared_error(y_test, y_pred, squared=False)
print("Root Mean Squared Error (RMSE):", rmse)

# Comment on prediction accuracy
```

```
Root Mean Squared Error (RMSE): 105189.89535216476
```

```
C:\Users\suneh\anaconda3\Lib\site-packages\sklearn\metrics\_regression.py:48
3: FutureWarning: 'squared' is deprecated in version 1.4 and will be removed
in 1.6. To calculate the root mean squared error, use the function'root_mean
_squared_error'.
  warnings.warn(
```

Predictions for New Properties

Given the following property features:

- Property 1: 678 sq ft, 1 room, Downtown, built in 2019
- Property 2: 1550 sq ft, 4 rooms, Suburb, built in 1972
- Property 3: 2509 sq ft, 3 rooms, Suburb, built in 2004

We predict their selling prices using the trained model.

In [79]:
```
# New property data
new_listings = pd.DataFrame({
    'Size (sq ft)': [678, 1550, 2509],
    'Number of Rooms': [1, 4, 3],
    'Neighborhood_Downtown': [1, 0, 0],
    'Neighborhood_Suburb': [0, 1, 1],
    'Age': [2024 - 2019, 2024 - 1972, 2024 - 2004]
})

new_listings.head()
```

Out[79]:

	Size (sq ft)	Number of Rooms	Neighborhood_Downtown	Neighborhood_Suburb	Age
0	678	1	1	0	5
1	1550	4	0	1	52
2	2509	3	0	1	20

In [81]:
```
new_listings.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 5 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Size (sq ft)           3 non-null      int64
 1   Number of Rooms        3 non-null      int64
 2   Neighborhood_Downtown  3 non-null      int64
 3   Neighborhood_Suburb    3 non-null      int64
 4   Age                    3 non-null      int64
dtypes: int64(5)
memory usage: 252.0 bytes
```

In [83]:
```
# Align columns with training data
new_listings = new_listings.reindex(columns=X.columns, fill_value=0)

# Predict prices for the new listings
predicted_prices = model.predict(new_listings)

# Display the predicted prices
print(predicted_prices)
```

```
[259131.26841892 559266.0919646  914604.9120119 ]
```

Import the Dataset and Describe Summary Statistics

This step involves:

1. Importing the `cancer_data.csv` dataset into a Pandas DataFrame.
2. Using the `.describe()` function to summarize the dataset's statistics.

In [97]:
```
import pandas as pd

# Import the dataset
data2 = pd.read_csv("C:/Users/suneh/Downloads/cancer_data (1).csv")

#Show data
data2.head()
```

Out[97]:

	Age	Smoking Status	Tumor Size (cm)	Cancer Stage	Treatment Type	Survived
0	64	Smoker	1.45	Stage I	Chemotherapy	1
1	67	Smoker	3.02	Stage II	Immunotherapy	1
2	84	Smoker	1.13	Stage I	Surgery	0
3	87	Smoker	1.12	Stage I	Radiation	1
4	87	Smoker	8.63	Stage IV	Radiation	0

In [99]:

```
# Display summary statistics
data2.describe()
```

Out[99]:

	Age	Tumor Size (cm)	Survived
count	500.000000	500.000000	500.000000
mean	54.836000	5.258220	0.488000
std	20.465355	2.800046	0.500357
min	20.000000	0.510000	0.000000
25%	38.000000	2.937500	0.000000
50%	55.000000	5.105000	0.000000
75%	73.000000	7.822500	1.000000
max	89.000000	10.000000	1.000000

One-Hot Encoding for Categorical Variables

The dataset contains several categorical variables (`Smoking Status` , `Cancer Stage` , `Treatment Type`). We'll apply one-hot encoding for each variable with `Number of Categories - 1` columns added.

In [109…

```
# One-hot encoding for categorical variables
categorical_vars2 = ['Smoking Status', 'Cancer Stage', 'Treatment Type']
data_encoded2 = pd.get_dummies(data2, columns=categorical_vars2, drop_first

# Display the first few rows of the encoded DataFrame
data_encoded2.head()
```

Out[109…

	Age	Tumor Size (cm)	Survived	Smoking Status_Smoker	Cancer Stage_Stage II	Cancer Stage_Stage III	Cancer Stage_Stage IV
0	64	1.45	1	1	0	0	0
1	67	3.02	1	1	1	0	0
2	84	1.13	0	1	0	0	0
3	87	1.12	1	1	0	0	0
4	87	8.63	0	1	0	0	1

Logistic Regression Model

We train a logistic regression model with the following steps:

1. Split the dataset into training and testing sets (80% train, 20% test, random state=18).
2. Train the model using the training set.
3. Display the model's intercept and coefficients.

In [117…

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Define features and target variable
X = data_encoded2.drop(columns=['Survived'])
y = data_encoded2['Survived']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra

# Train logistic regression model
log_reg2 = LogisticRegression(max_iter=1000)
log_reg2.fit(X_train, y_train)

# Display intercept and coefficients
print("Intercept:", log_reg2.intercept_)
print("Coefficients:", dict(zip(X.columns, log_reg2.coef_[0])))
```

```
Intercept: [4.87474078]
Coefficients: {'Age': -0.04177729994934706, 'Tumor Size (cm)': -0.5529144370
828158, 'Smoking Status_Smoker': 0.2097765311753643, 'Cancer Stage_Stage I
I': 0.3042544502070155, 'Cancer Stage_Stage III': 0.16277538462039728, 'Canc
er Stage_Stage IV': -0.15137750235594935, 'Treatment Type_Immunotherapy': 0.
2633742471457567, 'Treatment Type_Radiation': 0.20610678832857524, 'Treatmen
t Type_Surgery': -0.5095647174745114}
```

Model Evaluation

1. Generate the confusion matrix based on predictions from the logistic regression model.
2. Calculate:
   - Precision (Fraction of predicted survivors who actually survived).
   - Recall (Fraction of actual survivors who were correctly predicted).
   - Accuracy.

In [129…

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score

# Predictions on the test set
y_pred = log_reg2.predict(X_test)

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)

# Performance metrics
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"Accuracy: {accuracy}")
```

```
Confusion Matrix:
 [[42 11]
 [11 36]]
Precision: 0.7659574468085106
Recall: 0.7659574468085106
Accuracy: 0.78
```

Predict Survival for New Patients

Two new patients:

1. Age: 34, Smoking Status: Smoker, Tumor Size: 0.5 cm, Cancer Stage: Stage II, Treatment Type: Immunotherapy.
2. Age: 87, Smoking Status: Non-Smoker, Tumor Size: 5 cm, Cancer Stage: Stage I, Treatment Type: Surgery.

The data is pre-processed similarly before making predictions.

In [137…

```
# New patient data
new_patients = pd.DataFrame({
```

```
    'Age': [34, 87],
    'Tumor Size (cm)': [0.5, 5],
    'Smoking Status_Non-Smoker': [0, 1],
    'Cancer Stage_Stage II': [1, 0],
    'Cancer Stage_Stage III': [0, 0],
    'Treatment Type_Immunotherapy': [1, 0],
    'Treatment Type_Radiation': [0, 0],
    'Treatment Type_Surgery': [0, 1]
})

new_patients.head()
```

Out[137…

	Age	Tumor Size (cm)	Smoking Status_Non-Smoker	Cancer Stage_Stage II	Cancer Stage_Stage III	Treatment Type_Immunotherapy	Ty
0	34	0.5	0	1	0	1	
1	87	5.0	1	0	0	0	

In [139…

```
# Align columns with training data
new_patients = new_patients.reindex(columns=X.columns, fill_value=0)

# Predict probabilities
predicted_probs = log_reg2.predict_proba(new_patients)[:, 1]
print("Predicted Probabilities of Survival:\n", predicted_probs)
```

```
Predicted Probabilities of Survival:
 [0.97692103 0.11567693]
```

5-Nearest Neighbors (5NN)

We repeat steps c-e with the 5-Nearest Neighbors algorithm, comparing its performance with logistic regression.

In [143…

```
from sklearn.neighbors import KNeighborsClassifier

# Train 5NN model
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Predict and evaluate
y_pred_knn = knn.predict(X_test)
conf_matrix_knn = confusion_matrix(y_test, y_pred_knn)

precision_knn = precision_score(y_test, y_pred_knn)
recall_knn = recall_score(y_test, y_pred_knn)
accuracy_knn = accuracy_score(y_test, y_pred_knn)

print("Confusion Matrix (5NN):\n", conf_matrix_knn)
```

```
print(f"Precision (5NN): {precision_knn}")
print(f"Recall (5NN): {recall_knn}")
print(f"Accuracy (5NN): {accuracy_knn}")

# Predictions for new patients
predicted_probs_knn = knn.predict_proba(new_patients)[:, 1]
print("Predicted Probabilities of Survival (5NN):\n", predicted_probs_knn)
```

```
Confusion Matrix (5NN):
 [[40 13]
 [11 36]]
Precision (5NN): 0.7346938775510204
Recall (5NN): 0.7659574468085106
Accuracy (5NN): 0.76
Predicted Probabilities of Survival (5NN):
 [1. 0.]
```

Performance Comparison

Based on the provided metrics for Logistic Regression and 5NN, here's the comparison table: Metric Logistic Regression 5NN Precision 0.766 0.735 Recall 0.766 0.766 Accuracy 0.78 0.76

Analysis

Precision:

Logistic Regression performs slightly better (0.766) compared to 5NN (0.735). This means Logistic Regression is better at ensuring that patients predicted to survive 5 years actually do. Recall:

Both models have identical recall (0.766), meaning they are equally effective at identifying patients who actually survive. Accuracy:

Logistic Regression (0.78) has a slight edge over 5NN (0.76) in overall correct predictions. Complexity and Interpretability:

Logistic Regression is easier to interpret as it provides coefficients for each feature, indicating their impact on survival prediction. 5NN, on the other hand, is computationally more expensive, especially as the dataset size grows, since it calculates distances for each prediction.

Preferred Model

Logistic Regression is preferred because:

It has better precision and accuracy. It is computationally efficient and interpretable, making it suitable for understanding the relationship between features and survival.

```
In [ ]:
```