Certainly! Here's a structured outline for documenting a NY-housing model using Different models using python.

**1. import numpy as np**

* This line imports the numpy library, typically assigned the alias np.
* NumPy is a fundamental library for scientific computing in Python.
* It provides powerful functions for working with multidimensional arrays and matrices.
* Common uses include numerical computations, linear algebra operations, and random number generation.

**2. import pandas as pd**

* This line imports the pandas library, typically assigned the alias pd.
* Pandas is a high-level data structure library built on top of NumPy.
* It excels at working with tabular data, providing data structures like DataFrames (think spreadsheet) and Series (think single column).
* Pandas offers powerful tools for data cleaning, manipulation, analysis, and visualization.

**3. import matplotlib.pyplot as plt**

* This line imports the matplotlib.pyplot module from the matplotlib library, typically assigned the alias plt.
* Matplotlib is a popular library for creating static, animated, and interactive visualizations in Python.
* It offers a wide variety of plot types (scatter plots, line plots, histograms, etc.) and extensive customization options.

**4. import seaborn as sns**

* This line imports the seaborn library, typically assigned the alias sns.
* Seaborn is a high-level library built on top of Matplotlib, designed specifically for statistical data visualization.
* It provides a user-friendly interface for creating attractive and informative plots with minimal code.
* Seaborn offers a variety of pre-built plot types tailored for common data exploration tasks.

The code you provided, df = pd.read\_csv('/content/NY-House-Dataset.csv'), attempts to load data from a CSV file named "NY-House-Dataset.csv" into a Pandas DataFrame named df. Here's a breakdown of what the code does:

1. **import pandas as pd**: This line imports the pandas library and assigns it the alias pd. Pandas is a powerful library for data analysis in Python, especially for working with tabular data.
2. **df = pd.read\_csv('/content/NY-House-Dataset.csv')**: This line is the core of the code. It uses the pd.read\_csv function from the pandas library to read data from a CSV file. Let's break down this part further:
   * df =: Assigns the result of the pd.read\_csv function to a variable named df. This variable will now hold the data from the CSV file.
   * pd.read\_csv\*\*: This is a function from the pandas library used for reading data from CSV files.
   * '/content/NY-House-Dataset.csv'\*\*: This is the path to the CSV file that you want to read. In this specific case, the file is assumed to be located in a directory called/content/`.

**However, there are a few things to keep in mind:**

* **Filepath**: The code assumes the CSV file is located at /content/NY-House-Dataset.csv. This path might need adjustment depending on where the file is actually stored.
* **Data Availability**: If you are running this code outside of a platform like Google Colab that provides the file by default, you'll need to have the CSV file available in the specified location or adjust the path to point to where the file is stored on your system.

1- Descriptive statistics

The code df.head() is used with a Pandas DataFrame (df in this case) to view the first few rows of the data. Here's a breakdown of what it does:

**1. df**: This refers to the Pandas DataFrame that you previously created using pd.read\_csv to load data from a CSV file. It's assumed the DataFrame has data loaded and is ready for exploration.

**2. .head()**: This is a method attached to the DataFrame object (df). The .head() method, without any arguments, by default returns the first 5 rows of the data in the DataFrame.

**In essence, df.head() provides a quick glimpse at the beginning of your data to get a sense of the information it contains and the structure of the data (column names, data types).**

Here are some additional points to consider:

* **Customizing the number of rows**: You can optionally specify the number of rows to return within the .head() method. For example, df.head(10) would return the first 10 rows of the DataFrame.
* **Data exploration**: df.head() is a helpful tool during data exploration. By looking at the initial rows, you can get a basic understanding of the data's content, identify potential issues (missing values, incorrect data types), and guide further analysis steps.

The code df.shape is used with a Pandas DataFrame (df in this case) to get information about the dimensions of the data. Here's a breakdown of what it does:

**1. df**: This refers to the Pandas DataFrame that you previously created, likely using pd.read\_csv to load data from a CSV file. It's assumed the DataFrame has data loaded.

**2. .shape**: This is an attribute attached to the DataFrame object (df). The .shape attribute returns a tuple containing two integers representing the dimensions of the data in the DataFrame.

**The elements of the tuple returned by df.shape signify:**

* **The first element**: Represents the number of rows (observations) in the DataFrame.
* **The second element**: Represents the number of columns (features) in the DataFrame.

**In essence, df.shape provides a concise way to determine the size and structure of your data.**

For example, if df.shape returns (100, 5), it indicates that the DataFrame has 100 rows (observations) and 5 columns (features). This information is useful for various purposes:

* **Understanding data size**: It helps you grasp the overall volume of data you're working with.
* **Data manipulation**: Knowing the number of rows and columns is crucial for various data manipulation tasks using Pandas methods (e.g., selecting specific rows or columns).
* **Exploratory analysis**: It can be a starting point for exploring the data's structure and potential relationships between variables (columns).

The code df.isnull().sum() is used with a Pandas DataFrame (df in this case) to identify and count missing values in the data. Here's a breakdown of what it does:

**1. df**: This refers to the Pandas DataFrame that you previously created, likely using pd.read\_csv to load data from a CSV file. It's assumed the DataFrame has data loaded.

**2. .isnull()**: This method is applied to the DataFrame (df) and returns a DataFrame of the same shape, but with boolean values (True or False). True indicates locations where there are missing values in the original DataFrame, and False indicates valid values.

**3. .sum()**: This method is applied to the result of .isnull(), which is a DataFrame of booleans. By default, .sum() sums the values along the rows (axis=0). In this case, it calculates the sum of True values (missing values) in each column.

**In essence, df.isnull().sum() provides a series containing the count of missing values for each column in the DataFrame.**

Here are some additional points to consider:

* **Understanding missing data**: This code helps identify which columns have missing data and how extensive the issue might be in each column.
* **Data cleaning decisions**: The results can inform your data cleaning decisions. You might choose to impute missing values, drop rows/columns with excessive missingness, or address the issue in other ways depending on your analysis goals and the nature of the data.
* **Alternative axis**: By default, .sum() sums along the rows (axis=0). You can optionally specify axis=1 to sum the missing values by row instead, giving you the count of missing values in each row (observation).

The code df.duplicated() is used with a Pandas DataFrame (df in this case) to identify rows containing duplicate values. Here's a breakdown of what it does:

**1. df**: This refers to the Pandas DataFrame that you previously created, likely using pd.read\_csv to load data from a CSV file. It's assumed the DataFrame has data loaded.

**2. .duplicated()**: This method is applied to the DataFrame (df) and returns a Boolean Series with the same index as the DataFrame.

* A value of True in the Boolean Series indicates that the corresponding row in the DataFrame is a duplicate.
* A value of False indicates that the corresponding row is unique (not a duplicate).
* The code df.info() is used with a Pandas DataFrame (df in this case) to display a concise summary of the DataFrame's structure and information. Here's a breakdown of what it does:
* **1. df**: This refers to the Pandas DataFrame that you previously created using pd.read\_csv to load data from a CSV file. It's assumed the DataFrame has data loaded and is ready for exploration.
* **2. .info()**: This is a method attached to the DataFrame object (df). The .info() method prints a detailed summary of the DataFrame's characteristics to the console.

The code df.describe() is used with a Pandas DataFrame (df in this case) to generate descriptive statistics that summarize the central tendency, dispersion, and shape of the data's distribution. Here's a breakdown of what it does:

**1. df**: This refers to the Pandas DataFrame that you previously created using pd.read\_csv to load data from a CSV file. It's assumed the DataFrame has data loaded and is ready for analysis.

**2. .describe()**: This method is applied to the DataFrame (df) and returns a summary of descriptive statistics for the numerical columns in the DataFrame.

The code you provided utilizes two libraries, matplotlib.pyplot (imported as plt) and likely pandas (assumed to be imported as pd), to create and display a histogram. Here's a breakdown of what the code does:

**1. plt.hist(df['TYPE'],bins=50)**:

* plt.hist: This refers to the hist function from the matplotlib.pyplot library. This function is used to create a histogram visualization.
* df['TYPE']: This part specifies the data to be visualized. It extracts the column named 'TYPE' from the Pandas DataFrame df.
* bins=50: This is an optional argument to the hist function. It sets the number of bins to be used in the histogram. In this case, 50 bins are specified.

**In essence, this line of code creates a histogram visualization of the data in the 'TYPE' column of the DataFrame df using 50 bins.**

**2. plt.show()**:

* plt.show: This function, from matplotlib.pyplot, displays the plot that was created using the preceding function calls (in this case, the histogram).

A graph with blue lines

Description automatically generated

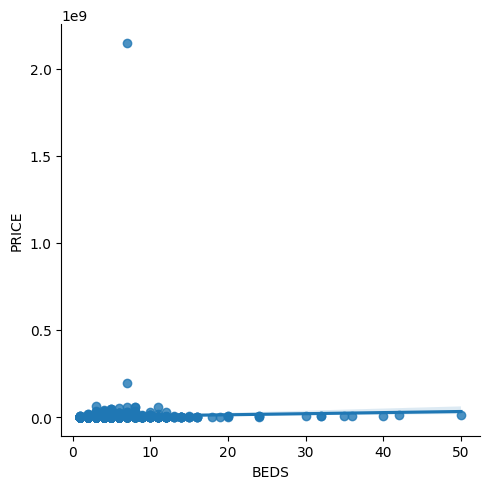
1. **seaborn (imported as sns)**: This library offers a high-level interface on top of matplotlib for creating statistical data visualizations.
2. **pandas (assumed to be imported as pd)**: This library is used for data analysis and manipulation, likely used to load your data into a DataFrame named df.

Here's a breakdown of what the code does:

* **sns.lmplot(x='BEDS', y='PRICE', data=df)**: This line creates a linear regression plot using the lmplot function from seaborn.
  + x='BEDS': This specifies the name of the column from the DataFrame (df) that provides the data for the x-axis. In this case, it's the 'BEDS' column.
  + y='PRICE': This specifies the name of the column from the DataFrame (df) that provides the data for the y-axis. In this case, it's the 'PRICE' column.
  + data=df: This argument indicates the DataFrame (df) that holds the data used to create the plot.

**In essence, this line creates a scatter plot showing the relationship between the number of beds (BEDS) and the price (PRICE) in your data, along with a fitted regression line.**

* **plt.show()**: This line, from matplotlib.pyplot, displays the plot that was crated using sns.lmplot.



**2- Hypothesis testing**

**from scipy.stats import shapiro**: This line imports the shapiro function from the scipy.stats library. The shapiro function is used to perform the Shapiro-Wilk test, a statistical test for normality.

**2. DataToTest = df['PRICE']**: This line extracts the 'PRICE' column from your DataFrame df and assigns it to a variable named DataToTest. This variable now holds the data you want to test for normality.

**3. stat, p = shapiro(DataToTest)**: This line applies the shapiro function to the data in DataToTest.

* stat: This variable stores the W-statistic, a test statistic calculated by the Shapiro-Wilk test. It reflects how closely the data resembles a normal distribution (higher values indicate greater similarity).
* p: This variable stores the p-value, which represents the probability of observing a W-statistic as extreme or more extreme than the one calculated, assuming the data is actually normally distributed.

**4. print('stat=%.2f, p=%.30f' % (stat, p))**: This line prints the calculated W-statistic (rounded to two decimal places) and the p-value (rounded to 30 decimal places) in a formatted string.

**5. if p > 0.05:**: This line starts a conditional statement. The condition checks if the p-value (p) is greater than 0.05.

* **print('Normal distribution')**: If the condition p > 0.05 is True (meaning the p-value is greater than 0.05), this line is executed. It prints a message indicating that the data appears to be normally distributed according to the Shapiro-Wilk test.
* **else:**: This block executes if the condition p > 0.05 is False (meaning the p-value is less than or equal to 0.05).
  + **print('Not a normal distribution')**: This line prints a message indicating that the data likely does not follow a normal distribution based on the Shapiro-Wilk test.

**Understanding the t-test:**

An independent samples t-test is a statistical test used to compare the means of two independent groups. It assumes the data in both groups is normally distributed. The test calculates a t-statistic and a p-value to assess if the observed difference between the means is likely due to random chance or if there's a statistically significant difference between the groups.

**Code Breakdown:**

1. **from scipy.stats import ttest\_ind**: This line imports the ttest\_ind function from the scipy.stats library. This function is specifically designed to perform independent samples t-tests.
2. **stat, p = ttest\_ind(df['BEDS'], df['PRICE'])**: This line applies the ttest\_ind function.
   * df['BEDS']: This represents the first group of data for the test. It extracts the 'BEDS' column from your DataFrame (df).
   * df['PRICE']: This represents the second group of data for the test. It extracts the 'PRICE' column from your DataFrame (df).
   * stat: This variable stores the t-statistic calculated by the test. It reflects the standardized difference between the means of the two groups.
   * p: This variable stores the p-value, which represents the probability of observing a t-statistic as extreme or more extreme than the one calculated, assuming the null hypothesis (no significant difference between the means) is true.

**Interpretation in the context of your data:**

Assuming you have a DataFrame df containing information about house listings, this code performs a t-test to compare the mean number of beds (df['BEDS']) between listings in two potentially different categories (represented by the separation of data into two groups). The test result is stored in the stat and p variables.

* **A statistically significant difference**: If the p-value (p) is less than a chosen significance level (commonly 0.05), we can reject the null hypothesis and conclude that there's a statistically significant difference between the average number of beds in the two groups. This could be between different property types, locations, or any other categorical variable that separates your data into two sets of listings.
* **No statistically significant difference**: If the p-value (p) is greater than or equal to the significance level, we fail to reject the null hypothesis. In this case, we cannot conclude with statistical certainty that there's a significant difference between the mean number of beds in the two groups based on the t-test.

**Understanding the One-Sample T-Test:**

A one-sample t-test is a statistical test used to compare the mean of a sample (df['BEDS'] in your case) to a hypothesized population mean (20 in your case). It assumes the data in the sample is normally distributed. The test calculates a t-statistic and a p-value to assess if the observed difference between the sample mean and the hypothesized mean is likely due to random chance or if there's a statistically significant difference.

**Code Breakdown:**

1. **import scipy**: This line imports the scipy library, which provides various scientific computing functions.
2. **scipy.stats.ttest\_1samp(df['BEDS'],20)**: This line applies the ttest\_1samp function from scipy.stats.
   * df['BEDS']: This represents the data for the test. It extracts the 'BEDS' column from your DataFrame (df). This is the sample you're comparing to the hypothesized mean.
   * 20: This is the hypothesized population mean. You're essentially asking if the average number of beds in your sample (df['BEDS']) is statistically different from 20.
   * The function returns a tuple containing two elements:
     + t-statistic: This reflects the standardized difference between the sample mean and the hypothesized mean.
     + p-value: This represents the probability of observing a t-statistic as extreme or more extreme than the one calculated, assuming the null hypothesis (no significant difference between the sample mean and the hypothesized population mean) is true.

**Interpretation in the context of your data:**

Assuming you have a DataFrame df containing information about house listings, this code performs a t-test to see if the average number of beds (in df['BEDS']) differs significantly from a hypothesized average of 20 beds. The test result is stored in the t-statistic and p-value returned by the function.

* **Statistically significant difference**: If the p-value is less than a chosen significance level (commonly 0.05), we can reject the null hypothesis and conclude that there's a statistically significant difference between the average number of beds in your sample and the hypothesized value of 20.
* **No statistically significant difference**: If the p-value is greater than or equal to the significance level, we fail to reject the null hypothesis. In this case, we cannot conclude with statistical certainty that there's a significant difference between the average number of beds in your sample and the hypothesized value of 20 based on the t-test.

**3- Linear regression:**

**1. DataFrames and Subsetting:**

* You likely have a Pandas DataFrame named df that stores your data. A DataFrame is a tabular data structure with labeled rows and columns.
* In this code, you're separating specific columns from the DataFrame.

**2. X = df[['BEDS','BATH']]**

* This line creates a new DataFrame named X.
* The [[]] notation is used for subsetting rows or columns in a DataFrame.
* In this case, you're providing a list of column names within the brackets: 'BEDS' and 'BATH'.
* This selects these two specific columns from the original DataFrame df and assigns them to the new DataFrame X. So, X will now only contain the 'BEDS' and 'BATH' columns from the original data.

**3. y = df['PRICE']**

* This line creates a new variable named y.
* It uses single square brackets [] to select a single column by name from the DataFrame.
* Here, you're extracting the 'PRICE' column from df and assigning it to the variable y. So, y will now be a Series (a one-dimensional labeled array) containing the price values from the original DataFrame.

**In essence, you're separating your data into two parts:**

* X: This DataFrame holds the feature variables, which are the 'BEDS' and 'BATH' columns that might potentially influence the price.
* y: This variable holds the target variable, which is the 'PRICE' column that you're trying to predict or analyze.

**from sklearn.model\_selection import train\_test\_split**: This line imports the train\_test\_split function from the sklearn.model\_selection module. This function is widely used for splitting data into training and testing sets in machine learning.

**2. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20, random\_state=42)**: This line splits the data you prepared earlier (X for features and y for target variable) into training and testing sets.

* X\_train, X\_test: These variables store the training and testing sets for the features (independent variables) from the original X DataFrame.
* y\_train, y\_test: These variables store the training and testing sets for the target variable (dependent variable) from the original y variable.
* test\_size=0.20: This argument specifies the proportion of data to be included in the testing set. In this case, 20% (0.20) of the data is allocated to the testing set, and the remaining 80% is used for training.
* random\_state=42: This argument helps ensure reproducibility by setting a seed for the random number generator used during the splitting process. This means if you run the code again with the same random\_state value, you'll get the same split of data into training and testing sets.

**3. print('Training Data Shape:', X\_train.shape)**: This line prints the shape (number of rows and columns) of the training data (X\_train).

**4. print('Testing Data Shape: ', X\_test.shape)**: This line prints the shape (number of rows and columns) of the testing data (X\_test).

**In essence, this code separates your data into two crucial sets:**

* **Training data (X\_train, y\_train)**: This larger portion of the data is used to train a machine learning model. The model learns patterns and relationships between the features (X\_train) and the target variable (y\_train).
* **Testing data (X\_test, y\_test)**: This is a held-out set of data that the model has never seen before. It's used to evaluate the generalizability and performance of the trained model on unseen data.
* y\_train likely holds a Pandas Series or NumPy array containing the target variable values for your training data. This target variable is what you're trying to predict or analyze (e.g., housing prices, customer churn, etc.).

**value\_counts() Method:**

* The .value\_counts() method is applied to y\_train. This method is commonly used with Pandas Series or NumPy arrays containing categorical data.
* When applied, it calculates the frequency of each unique value in the Series/array.

**Output:**

* The output of y\_train.value\_counts() will be a new Series or array that shows how many times each unique value appears in y\_train. This essentially provides a count of each category (or class) present in your training target variable.

y\_test.value\_counts() works similarly to y\_train.value\_counts(), but applied to the testing data. Here's a breakdown of what it does and its significance in machine learning:

**Understanding y\_test:**

* Assuming you've split your data into training and testing sets using train\_test\_split, y\_test represents the target variable for your testing data. This data has not been used to train any model yet.

**value\_counts() Method:**

* The .value\_counts() method is applied to y\_test. This method is commonly used with Pandas Series or NumPy arrays containing categorical data.
* When applied, it calculates the frequency of each unique value in the Series/array.

**Output:**

* The output of y\_test.value\_counts() will be a new Series or array that shows how many times each unique value appears in y\_test. This essentially provides a count of each category (or class) present in your testing target variable.

1. **sklearn.model\_selection**: This sub-library of scikit-learn offers various functions for splitting data into training and testing sets, essential for machine learning tasks. Here, you're likely interested in the train\_test\_split function for this purpose.
2. **sklearn.linear\_model**: This sub-library provides various machine learning algorithms for linear regression problems. In this case, you're importing the LinearRegression class, which is used to fit a linear model to your data.

By combining these imports, you're laying the groundwork for building and evaluating a linear regression model using scikit-learn. Here's a general workflow you might follow:

1. **Import libraries:** You've already done this step by importing train\_test\_split from sklearn.model\_selection and LinearRegression from sklearn.linear\_model.
2. **Prepare data:** Load your data into a DataFrame and handle any missing values or data cleaning steps.
3. **Split data:** Use train\_test\_split to create training and testing sets for your features (independent variables) and target variable (dependent variable).
4. **Create the model:** Initialize an instance of the LinearRegression class.
5. **Train the model:** Fit the model to the training data using the fit method of the LinearRegression object. This involves the model learning the coefficients of the linear equation that best represents the relationship between the features and the target variable in your training data.
6. **Make predictions:** Use the trained model to predict the target variable for the unseen testing data using the predict method.
7. **Evaluate the model:** Assess the performance of the model on the testing data using various metrics (e.g., mean squared error, R-squared).

**1. regressor = LinearRegression()**

* This line creates an instance of the LinearRegression class from scikit-learn's linear\_model sub-library (assuming you've already imported it using from sklearn.linear\_model import LinearRegression).
* This class represents a linear regression model that you can train to fit a linear equation to your data.

**2. regressor.fit(X\_train, y\_train)**

* This line trains the linear regression model you just created (regressor).
* .fit(X\_train, y\_train) is the method used for training the model.
  + X\_train: This is the training data for the features (independent variables). It's likely a DataFrame or NumPy array containing the features you want to use for prediction.
  + y\_train: This is the training data for the target variable (dependent variable). It's likely a Pandas Series or NumPy array containing the values you want to predict.

**train\_score = regressor.score(X\_train, y\_train)**

* This line calculates the R-squared score of the trained linear regression model (regressor).
* .score(X\_train, y\_train) is a method provided by the LinearRegression class that computes the R-squared score between the predicted target variable values and the actual target variable values for the data provided.
* In this case, you're providing the training data (X\_train and y\_train) to calculate the score on the training set.

**2. print(f'Model Score/Performance on Training data: {round((train\_score\*10000), 2)}')**

* This line prints a message that includes the calculated R-squared score.
* It uses f-strings (introduced in Python 3.6) to embed the train\_score variable directly into the message.
* The score is multiplied by 10000 and rounded to two decimal places using round((train\_score\*10000), 2). This is likely to present the score as a percentage, as R-squared is typically between 0 and 1.

**test\_score = regressor.score(X\_test, y\_test)**

* This line calculates the R-squared score of the trained linear regression model (regressor) on the testing data.
* .score(X\_test, y\_test) is the same method used previously, but this time you're providing the testing data (X\_test and y\_test) to assess the model's performance on unseen data. Remember, the model was not trained on this data.

**2. print(f'Model Score/Performance on Testing data: {round((test\_score\*100), 2)}')**

* This line prints a message that includes the calculated R-squared score for the testing data.
* It uses f-strings to embed the test\_score variable into the message.
* The score is multiplied by 100 and rounded to two decimal places using round((test\_score\*100), 2). This is likely to present the score as a percentage, as R-squared is typically between 0 and 1.