Shweta_Shinde_Math_Lab1

October 26, 2024

0.0.1 Shweta Ajay Shinde 017548687

0.0.2 MSDA, SJSU , Data 220- Math Method for DA¶

Access the given dataset using panda's command 'read_csv'

[3]:
```
import pandas as pd

heart=pd.read_csv('F:\heart.csv')
heart.head()
```

[3]:
```
     sbp  tobacco   ldl  adiposity  famhist  typea  obesity  alcohol   age  \
0  134.0    13.60  3.50      27.78  Present   60.0    25.99    57.34  49.0
1  132.0     6.20  6.47      36.21  Present   62.0    30.77    14.14  45.0
2  142.0     4.05  3.38      16.20   Absent    NaN    20.81     2.62  38.0
3  114.0     4.08  4.59      14.60  Present   62.0    23.11     6.72  58.0
4  114.0      NaN  3.83      19.40  Present   49.0    24.86     2.49   NaN

   chd
0  1.0
1  0.0
2  0.0
3  NaN
4  NaN
```

1 Part1

Q1. Look at each variables(features) in the dataset and decide if it belongs to one of these four groups:

Based on the given data below are the observations.

1. **Nominal Categorical:** Categories with no specific order (like colors or names).
   - **famhist**: Family history (data is present/absent, but no order)

2. **Binary Categorical:** Two possible categories (like yes/no or 0/1).
   - **chd**: Coronary heart disease (0=Alive, 1=Dead)

- **famhist**: Could also fit here (0=Absent, 1=Present)

3. **Discrete:** Whole numbers (like number of items).
   - **typea**: Type-A behavior score (e.g., 49, 72, 65, 62)
   - **age**: Subject's age (e.g., 15, 17, 50, 49)
   - **sbp**: Systolic blood pressure (e.g., 122, 134, 114, 126)

4. **Continuous:** Numbers that can take any value within a range (like height or temperature).
   - **sbp**: Systolic blood pressure
   - **tobacco**: Cumulative tobacco consumption (kg)
   - **ldl**: Cholesterol level
   - **adiposity**: Adipose tissue concentration
   - **typea**: Type-A behavior score
   - **obesity**: Obesity level
   - **alcohol**: Alcohol consumption
   - **age**: Subject's age

Q2. Find the number of null values for each column. To find the number of null values in each column, we can use the **.isnull().sum()**.

- **isnull()** command will chcek if records in every column if is null and gives binary value 0(No) or 1(Yes).
- **sum()** will count 1 to get the number of nulls present

[3]:
```
null_values = heart.isnull().sum()
null_values
```

[3]:
```
sbp          28
tobacco      40
ldl          39
adiposity    40
famhist      45
typea        41
obesity      40
alcohol      40
age          35
chd          39
dtype: int64
```

3. Descriptive Analysis:

1) Show the general descriptive statistics by using describe function. **describe()** function gives data statistics like mean, median, std, etc.

[4]:
```
heart.describe()
```

[4]:

	sbp	tobacco	ldl	adiposity	typea	obesity
count	384.000000	372.000000	373.000000	372.000000	371.000000	372.000000
mean	139.216146	3.676425	4.569303	25.210753	52.008086	25.763602
std	20.307368	4.568564	1.888691	7.760257	9.822888	3.854265
min	101.000000	0.000000	0.980000	7.120000	20.000000	17.890000
25%	124.000000	0.057500	3.240000	19.307500	46.000000	22.835000
50%	136.000000	1.800000	4.220000	26.115000	52.000000	25.675000
75%	148.500000	5.640000	5.470000	30.790000	58.000000	28.167500
max	218.000000	27.400000	14.160000	42.490000	73.000000	40.340000

	alcohol	age	chd
count	372.000000	377.000000	373.000000
mean	18.425134	42.453581	0.335121
std	25.971090	15.312649	0.472667
min	0.000000	15.000000	0.000000
25%	0.195000	30.000000	0.000000
50%	7.300000	45.000000	0.000000
75%	25.820000	57.000000	1.000000
max	145.290000	64.000000	1.000000

2) Find the age of the oldest person and print the people with that age by filtering We use **max()** on the age column to find the maximum age, then filter the dataset where age equals that value.

[4]:
```
max_age=heart['age'].max()
print(f"Oldest persons ages is: {max_age}")
oldest_person=heart[heart['age']==max_age]
oldest_person
```

```
Oldest persons ages is: 64.0
```

[4]:

	sbp	tobacco	ldl	adiposity	famhist	typea	obesity	alcohol	age
58	158.0	3.60	2.97	NaN	Absent	NaN	26.64	108.00	64.0
70	152.0	12.18	4.04	37.83	Present	63.0	34.57	4.17	64.0
110	126.0	0.00	5.98	29.06	Present	56.0	25.39	11.52	64.0
167	148.0	8.20	7.75	34.46	Present	46.0	26.53	6.04	64.0
170	128.0	5.16	4.90	NaN	Present	57.0	26.42	0.00	64.0
206	NaN	8.60	3.90	32.16	Present	52.0	28.51	11.11	64.0
241	160.0	0.60	6.94	30.53	Absent	36.0	25.68	1.42	64.0
256	138.0	2.00	5.11	31.40	Present	49.0	27.25	2.06	64.0
276	128.0	0.73	3.97	23.52	Absent	NaN	23.81	NaN	64.0
348	140.0	8.60	3.90	32.16	Present	52.0	28.51	11.11	64.0
374	160.0	0.60	6.94	30.53	Absent	36.0	25.68	NaN	64.0

```
402  174.0     2.02  6.57      31.90  Present   50.0     28.75     11.83  64.0

     chd
58   0.0
70   0.0
110  1.0
167  1.0
170  0.0
206  1.0
241  0.0
256  1.0
276  0.0
348  1.0
374  0.0
402  1.0
```

3) Find the youngest person and print the people with that age by filtering We use **min()** on the age column to find the mainimum age, then filter the dataset where age equals that value.

[6]:
```python
min_age=heart['age'].min()
print(f"Yongest persons ages is: {min_age}")
youngest_person=heart[heart['age']==min_age]
youngest_person
```

```
Yongest persons ages is: 15.0
```

[6]:
```
      sbp  tobacco   ldl  adiposity famhist  typea  obesity  alcohol   age  \
9   132.0      0.0  1.87      17.21  Absent   49.0    23.63     0.97  15.0
38    NaN      0.0  3.67      12.13  Absent    NaN    19.15     0.60  15.0

    chd
9   0.0
38  0.0
```

4) Find the average and standard deviation of age column We calculate this using **mean()** and **std()** on the age column.

[7]:
```python
average_age = heart['age'].mean()
std_age = heart['age'].std()
print(f"Average age: {average_age:.4f}, Standard deviation: {std_age:.4f}")
```

```
Average age: 42.4536, Standard deviation: 15.3126
```

5)Find median age We use the **median()** function on the age column.

[8]:
```python
median_age=heart['age'].median()
print(f"Median age: {median_age}")
```

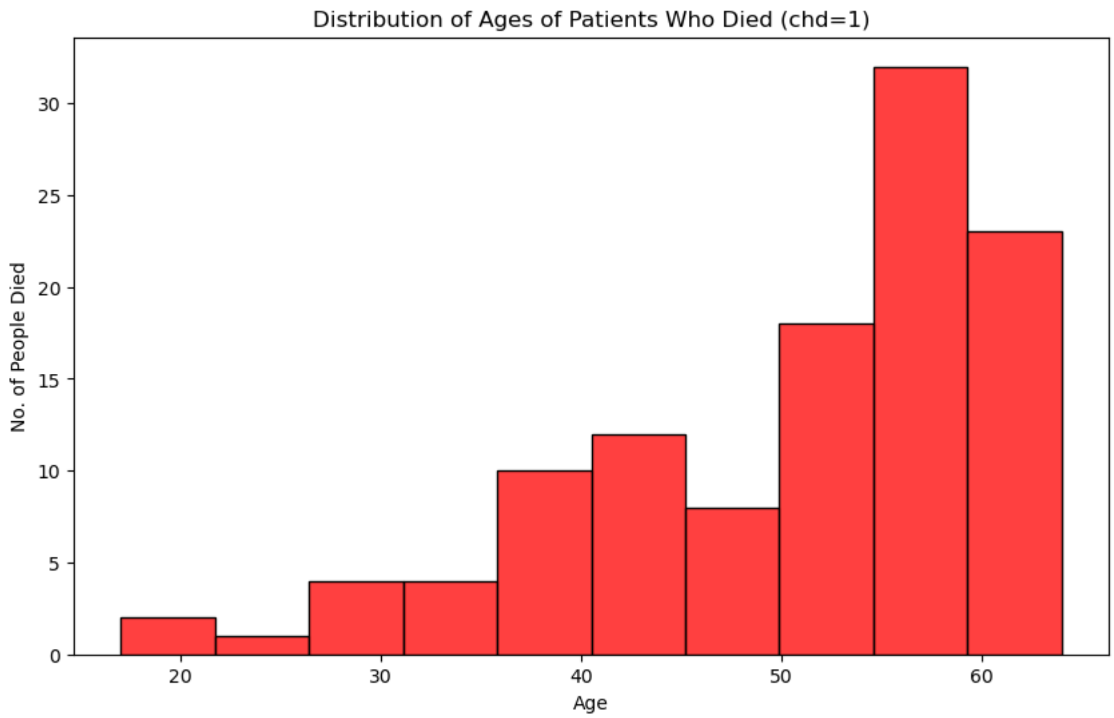
Median age: 45.0

6) Draw a bar chart that represents the relationship between the deaths and ages and draw an insights from the chart (you can filter by chd==1 and draw a histogram) We first filter the data where chd == 1, then plot a histogram of ages using a library like matplotlib or seaborn.

[2]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Filter data where chd == 1 (Deaths)
deaths = heart[heart['chd'] == 1]

# Draw a histogram of ages for those who died
plt.figure(figsize=(10, 6))
sns.histplot(deaths['age'], bins=10,color='red')
plt.title('Distribution of Ages of Patients Who Died (chd=1)')
plt.xlabel('Age')
plt.ylabel('No. of People Died')
plt.show()
```



Insights from the above Graph:

- More than 30 individuals died between the ages of 55 and 60.

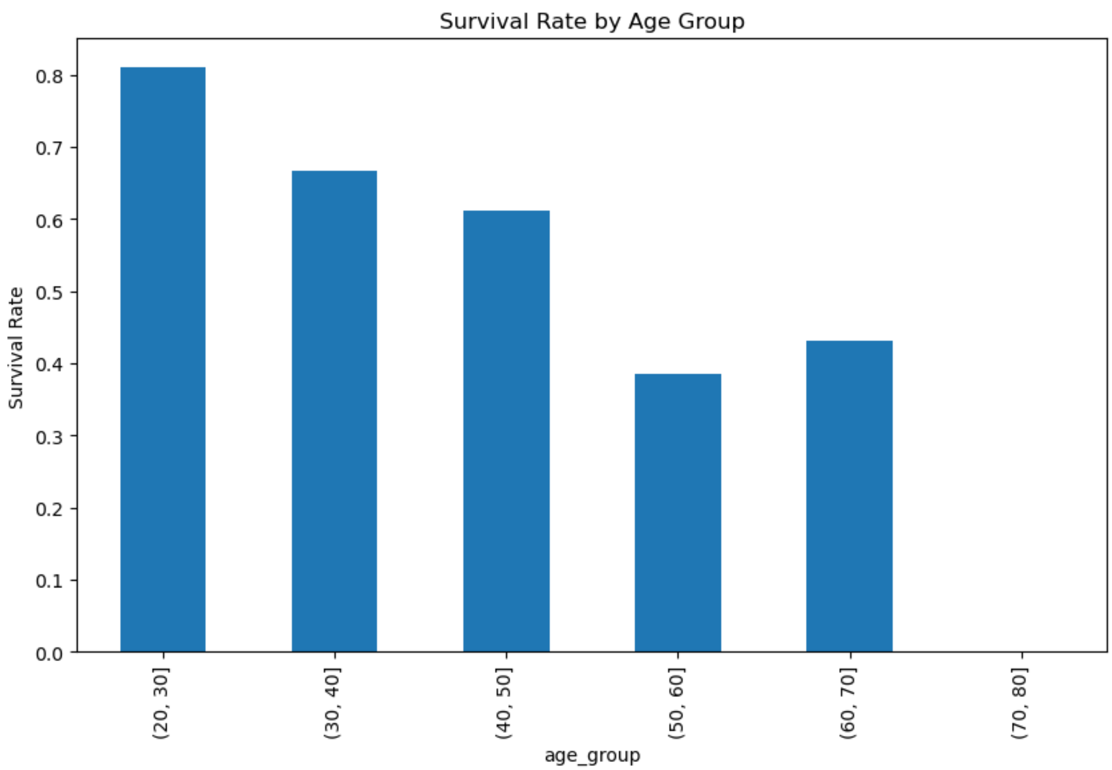
- Between 5 to 10 individuals died at a younger age, specifically between 15 and 35.
- The data shows a noticeable increase in mortality rates as age increases.
- CHD(Coronary heart disease) underlying health factors contributing to the higher mortality rates in older age groups.

7) Find the age groups whose survival rate is the largest (Please include a graph that proves it)

[10]:
```
# Creating age groups
bins = [20, 30, 40, 50, 60, 70, 80]
heart['age_group'] = pd.cut(heart['age'], bins)

# Calculate survival rate by age group (chd == 0 is survival)
survival_rate_by_age = heart[heart['chd'] == 0].groupby('age_group').size() /
    heart.groupby('age_group').size()

# Plot survival rates
survival_rate_by_age.plot(kind='bar', figsize=(10, 6), title='Survival Rate by
    Age Group')
plt.ylabel('Survival Rate')
plt.show()
```



Insights from the above Graph:

- People aged 20 to 30 have the highest survival rate of 80%(0.8), meaning younger people tend to stay healthier.
- People aged 50 to 60 have the lowest survival rate of 45%(0.45), likely because of more health problems as they get older.
- This shows how important it is to take care of your health, especially as you get older, to avoid serious issues.

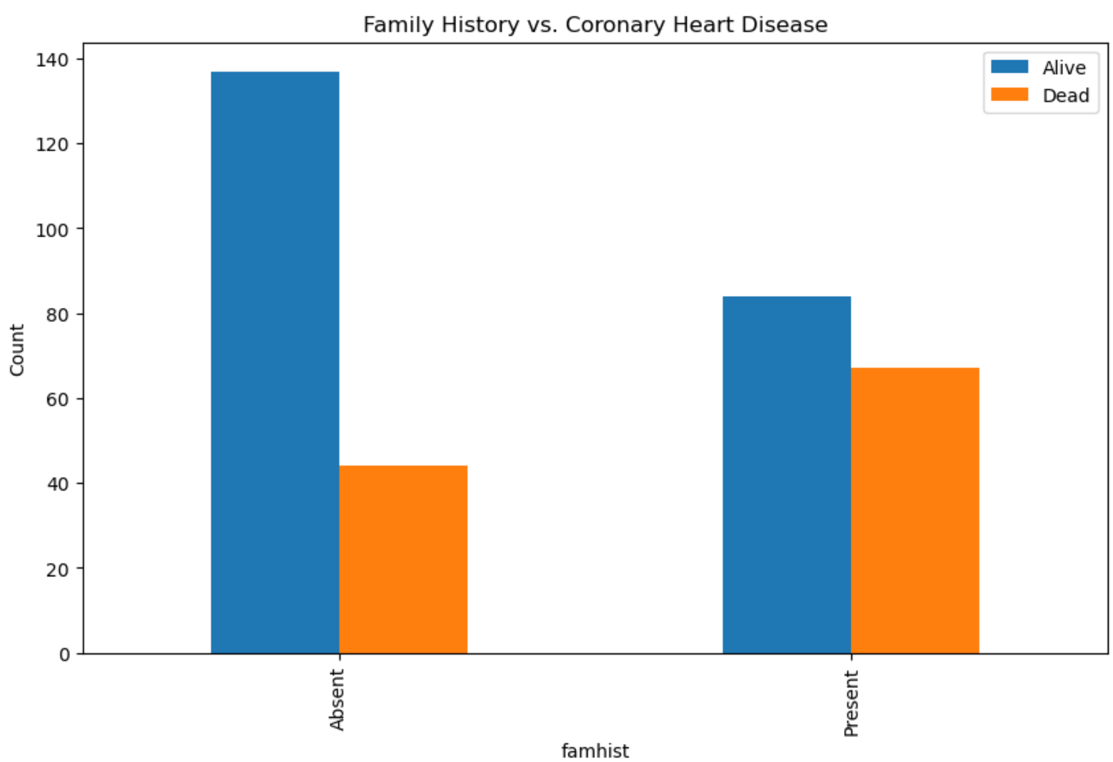
(20,30] means - Open interval (20): Ages greater than 20 are included, but 20 itself is not included. - Closed interval [30]: Ages up to and including 30 are included in this bin.

8) Find a relationship between 'famhist' and 'chd'. Please write what you have found.

[11]:
```
# Create a table
famhist_chd = pd.crosstab(heart['famhist'], heart['chd'])

# Rename the index of the chd variable for better clarity
famhist_chd.columns = ['Alive', 'Dead']

# Visualize the  table
famhist_chd.plot(kind='bar', figsize=(10, 6), title='Family History vs.␣
↪Coronary Heart Disease')
plt.ylabel('Count')
plt.show()
```



Insights from the above Graph:

- Most people survive when they don't have a family history of coronary heart disease (CHD).
- Fewer people survive if they do have a family history of CHD, showing that genetics can play a role.
- This shows the importance of regular check-ups and staying healthy, particularly for people with a family history of heart disease.

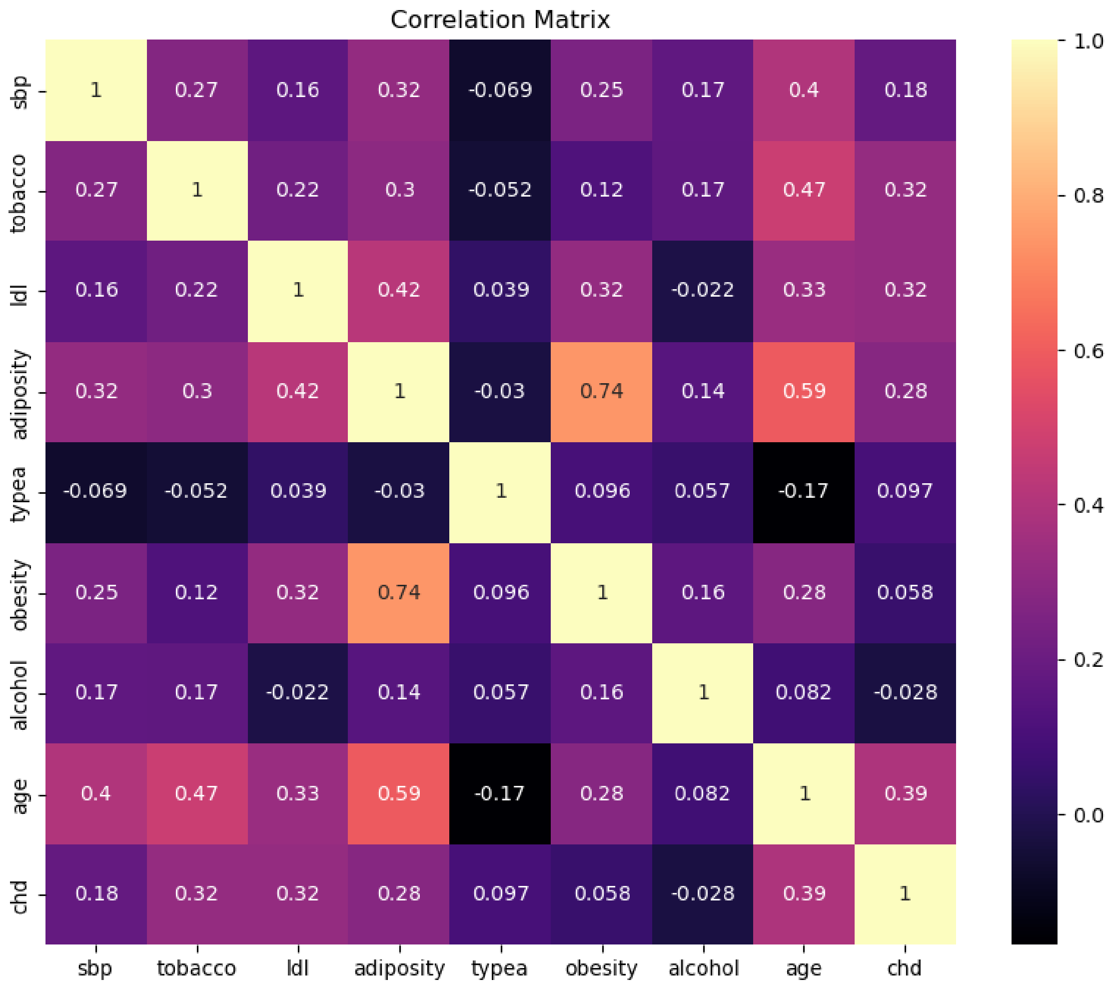
9) Get more visuals on data distributions

i. Plot Correlation Matrix We use .corr() and plot it using seaborn's heatmap().

Heat map is the best map to show corelations. Correlation measures how two variables are related. A positive correlation means both variables increase together, while a negative correlation means one increases as the other decreases. If the correlation is close to zero, the variables are not related. It's used to find patterns, make predictions, and understand relationships between data points. The strength of this relationship is shown by values ranging from +1 (strong positive) to -1 (strong negative).

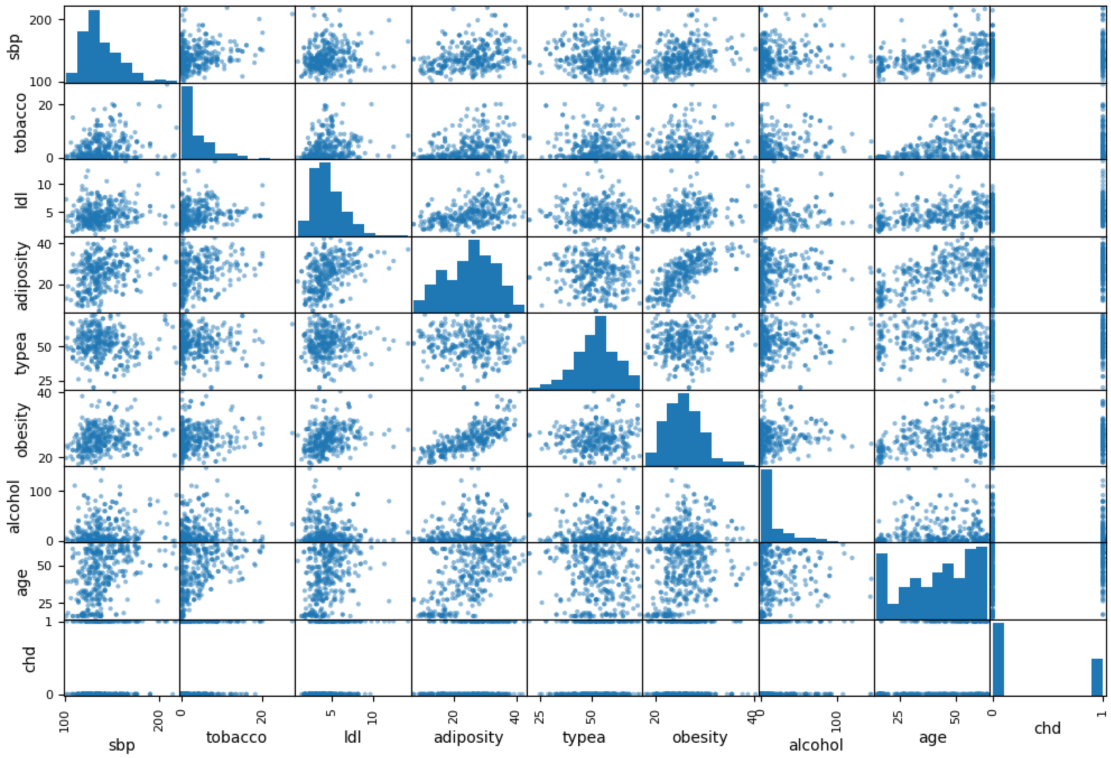
[20]:

```
# Plot correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(heart.corr(numeric_only=True), annot=True, cmap='magma')
plt.title('Correlation Matrix')
plt.show()
```



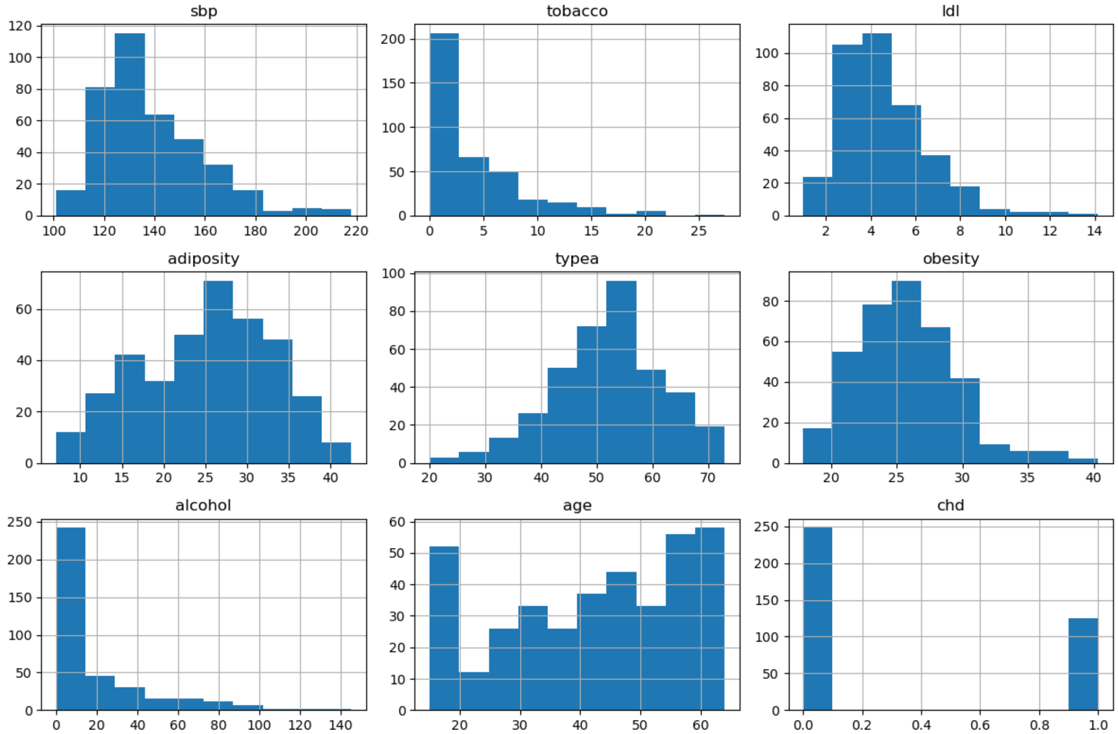
ii. Plot Scatter Matrix A scatter matrix is a grid of scatter plots that shows the relationships between multiple pairs of variables in a dataset. Each plot in the matrix represents the relationship between two variables, making it easy to spot patterns, correlations, and potential outliers across all variables at once.

[26]:
```
from pandas.plotting import scatter_matrix
# Scatter matrix
scatter_matrix(heart, figsize=(12, 8))
plt.show()
```



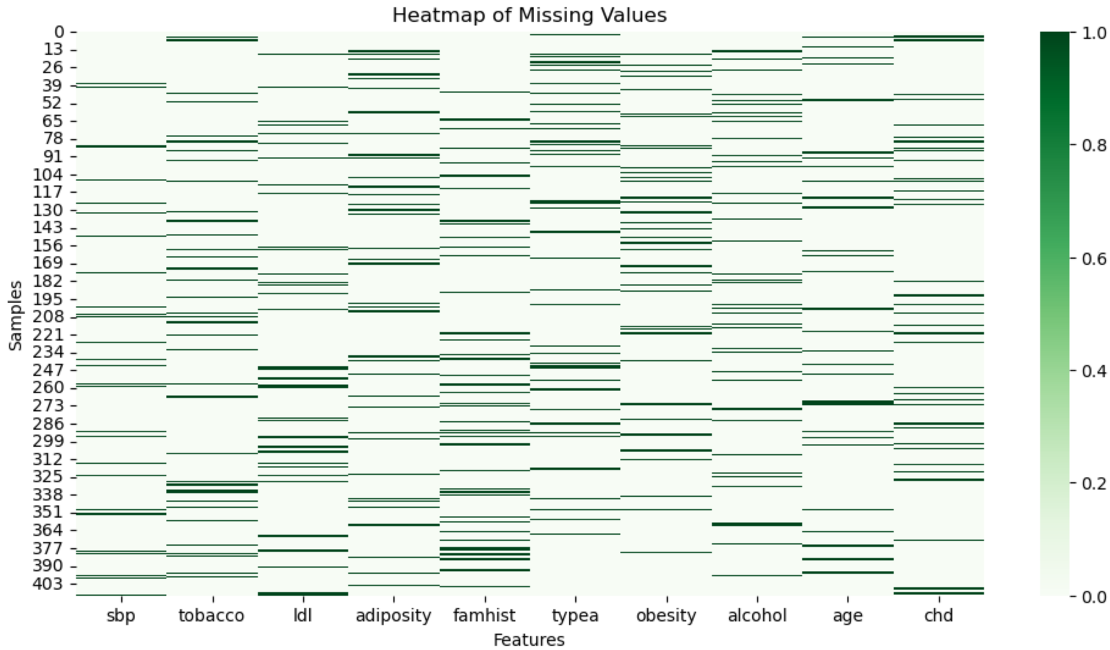
iii. Plot Per Column Distribution Plot per column distribution refers to visualizing the frequency or probability distribution of values for each column in a dataset. This can help identify patterns, trends, and potential outliers for each individual variable.

[31]:
```
heart.hist(figsize=(12, 8))
plt.tight_layout()
plt.show()
```



iiii. Plot a heat map for missing values

[6]:
```
missing_data = heart.isnull()
plt.figure(figsize=(12, 6))  # Increase figure size for better clarity
sns.heatmap(missing_data, cmap='Greens', cbar=True)  # Use gray for clarity
plt.title('Heatmap of Missing Values')
plt.xlabel('Features')
plt.ylabel('Samples')
plt.show()
```



10) Please research and provide your opinion on additional techniques for handling null values,excluding the 'drop NA' feature.

- Fill in missing values using the average (mean), middle value (median), or most common value (mode) from the data.
- Neighbor-Based Filling,Use similar data points to predict and fill in the missing values.
- Create a model that guesses the missing values based on the other information in the data.
- For data that changes over time, use methods to estimate missing values based on nearby points.
- Create fake data points to replace or fill in the missing information.

Below I am using fillna to fill missing values with mean() values

[14]:
```
heart.fillna(heart.mean(numeric_only=True),inplace=False)
```

[14]:
```
            sbp    tobacco       ldl  adiposity  famhist      typea  obesity  \
0    134.000000  13.600000  3.500000      27.78  Present  60.000000    25.99
1    132.000000   6.200000  6.470000      36.21  Present  62.000000    30.77
2    142.000000   4.050000  3.380000      16.20   Absent  52.008086    20.81
3    114.000000   4.080000  4.590000      14.60  Present  62.000000    23.11
4    114.000000   3.676425  3.830000      19.40  Present  49.000000    24.86
..          …          …        …     …        …        …        …
407  146.000000   3.600000  3.510000      22.67   Absent  51.000000    22.29
408  206.000000   0.000000  4.170000      33.23   Absent  69.000000    27.36
409  134.000000   3.000000  3.170000      17.91   Absent  35.000000    26.37
410  148.000000  15.000000  4.569303      36.94  Present  72.000000    31.83
```

```
411  139.216146   0.210000  4.569303      15.11   Absent  61.000000    22.17

     alcohol        age       chd
0      57.34  49.000000  1.000000
1      14.14  45.000000  0.000000
2       2.62  38.000000  0.000000
3       6.72  58.000000  0.335121
4       2.49  42.453581  0.335121
..       …         …        …
407    43.71  42.000000  0.335121
408     6.17  50.000000  1.000000
409    15.12  27.000000  0.000000
410    66.27  41.000000  0.335121
411     2.42  17.000000  0.000000

[412 rows x 10 columns]
```

2 Part 2

Write a Python function that takes two matrices as input and returns their product. Do not use built-in matrix multiplication functions such as np.dot.

A = [12 21] [2 8]

B = [13 7] [7, 8]

C=A×B

Calculating each element:

C[0,0]=12×13+21×7=156+147=303

C[0,1]=12×7+21×8=84+168=252

C[1,0]=2×13+8×7=26+56=82

C[1,1]=2×7+8×8=14+64=78

[3]:
```python
import numpy as np
def matrix_multiply(A, B):
    """
    Multiplies two matrices A and B.
    Parameters:
    A (numpy.ndarray): First matrix.
    B (numpy.ndarray): Second matrix.
    Returns:
    numpy.ndarray: The product of matrices A and B.
    """
    #1st we will get the shape of the given matrix and store in a each valiable
 ↪the size of rows and columns
    row_A, col_A = A.shape
```

```
    row_B, col_B = B.shape

    # Check if matrices are of correct size to multiply column A= Rowb
    if col_A != row_B:
        raise ValueError("Number of columns in A must equal the number of rows␣
↪in B.")

    #Create a result 0-matrix with same size to store the result
    result = np.zeros((row_A, col_B))

    # matrix multiplication
    for i in range(row_A):
        for j in range(col_B):
            for k in range(col_A):
                result[i, j] += A[i, k] * B[k, j]

    return result


A = np.array([[12, 21], [2, 8]])
B = np.array([[13, 7], [7, 8]])
print(matrix_multiply(A, B))
```

```
[[303. 252.]
 [ 82.  78.]]
```

Another method we can use is A@B as taught in class @ operator does multiplies 2 matrixes togther

[4]:
```
import numpy as np
def matrix_multiply(A, B):
    """
    Multiplies two matrices A and B.
    Parameters:
    A (numpy.ndarray): First matrix.
    B (numpy.ndarray): Second matrix.
    Returns:
    numpy.ndarray: The product of matrices A and B.
    """
    return A@B

A = np.array([[12, 21], [2, 8]])
B = np.array([[13, 7], [7, 8]])
print(matrix_multiply(A, B))
```

```
[[303 252]
 [ 82  78]]
```

2. Compute the Determinant Write a Python function that computes the determinant of a square matrix using the numpy.linalg module. For below matrix A determinant

A=[a b] [c d]

det(A)=(a×d)−(b×c)

[36]:
```
def compute_determinant(A):
    """
    Computes the determinant of a square matrix A.
    Parameters:
    A (numpy.ndarray): Square matrix.
    Returns:
    float: Determinant of the matrix A.
    """
    return np.linalg.det(A)


# Example usage:
A = np.array([[11, 13], [15, 17]])
print(compute_determinant(A))
```

```
-8.000000000000012
```

3)Solve a System of Linear Equations (10 pts) Write a function that solves a system of linear equations given in matrix form, Ax = b, where A is a coefficient matrix and b is a constant vector. Use numpy.linalg.solve(). A=[3 1] b=[9] = [3 1|9] [1 2] [8] [1 2|8]

- eq1:3x+1y=9 and eq2:1x+2y=8
- eq2: x=8-2y

substitute x value in eq1 3(8-2y)+1y=9

24-6y+1y=9

24-5y=9

-5y=9-24

-5y=-15 so, y=3

substitute y in eq2:

x=8-2y

x=8-2(3)

x=6

so we get [2,3]

[40]:
```
def solve_linear_system(A, b):
    """
    Solves the system of linear equations Ax = b.
```

```
    Parameters:
    A (numpy.ndarray): Coefficient matrix.
    b (numpy.ndarray): Constant vector.
    Returns:
    numpy.ndarray: Solution vector x.
    """
    return np.linalg.solve(A, b)


# Example usage:
A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])
print(solve_linear_system(A, b))
```

```
[2. 3.]
```

[ ]:

```

```