Metadata

```
Course:  DS 5100
Module:  06 Pandas
Topic:   HW Myocardial Infarction Analytics with Pandas
Author:  R.C. Alvarado (adapted)
Date:    7 July 2023
```

Student Info

- Name: Luke Schneider
- Net UD: vrd9sd
- URL of this file in GitHub: https://github.com/lukeschneider7/DS5100-vrd9sd/blob/main/lessons/M06/hw06.ipynb

Instructions

In your **private course repo on Rivanna**, use this Jupyter notebook and the data file described to write code that performs the tasks below.

Save your notebook in the `M06` directory.

Remember to add and commit these files to your repo.

Then push your commits to your repo on GitHib.

Be sure to fill out the **Student Info** block above.

To submit your homework, save the notebook as a PDF and upload it to GradeScope, following the instructions.

TOTAL POINTS: 12

Overview

In this homework, you will be working with the Myocardial Infarction (MI) Complications Data Set housed at UCI.

A myocardial infarction is commonly called a heart attack.

You may Read about the dataset in the Data Description File (DDF).

You will work with some of the columns (aka features).

A subset of these could be predictors in an ML model, while others could be outcome variables.

The section **Attribute Information** in the DDF provides details.

Setting Up

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

Prepare the Data

Read in the dataset from the UCI Machine Learning Repository.

Use Pandas' `read_csv()` function, giving the path to the dataset as an argument.

There is no header in this data, so pass a second argument `header=None`.

In [8]:
```
path_to_data = "http://archive.ics.uci.edu/ml/machine-learning-databases/005
```

Task 1

(1 PT)

Import the data into a datafram and then print the number of records in the dataset

In [10]:
```
# CODE HERE
# Task 1 - Answer
data = pd.read_csv(path_to_data, header=None)
print(data.head())
print(f'Number of Records: {len(data)}')
```

```
   0   1    2   3   4   5   6   7   8   9    ... 114 115 116 117 118 119 120  \
0    1  77    1   2   1   1   2   ?   3   0  ...   0   0   0   0   0   0   0
1    2  55    1   1   0   0   0   0   0   0  ...   0   0   0   0   0   0   0
2    3  52    1   0   0   0   2   ?   2   0  ...   0   0   0   0   0   0   0
3    4  68    0   0   0   0   2   ?   2   0  ...   0   0   0   0   0   0   1
4    5  60    1   0   0   0   2   ?   3   0  ...   0   0   0   0   0   0   0

  121 122 123
0   0   0   0
1   0   0   0
2   0   0   0
3   0   0   0
4   0   0   0

[5 rows x 124 columns]
Number of Records: 1700
```

Task 2

(1 PT)

Show the first three records in the dataset

In [12]:
```
# CODE HERE
# Task 2 - Response
print(data.head(3)) # Show the first three records in the dataset
```

```
   0   1    2   3   4   5   6   7   8   9    ... 114 115 116 117 118 119 120  \
0    1  77    1   2   1   1   2   ?   3   0  ...   0   0   0   0   0   0   0
1    2  55    1   1   0   0   0   0   0   0  ...   0   0   0   0   0   0   0
2    3  52    1   0   0   0   2   ?   2   0  ...   0   0   0   0   0   0   0

  121 122 123
0   0   0   0
1   0   0   0
2   0   0   0

[3 rows x 124 columns]
```

Working with AGE

The second column contains patient age.

If your dataframe is named `df`, you can reference the column with: `df[1]`.

Generally the field names will be strings and you can use `df['age']` to access field `age`, as an example).

Task 3

(1 PT)

One complication: missing values are filled with `?` which will cause problems (e.g., stats can't be computed easily).

Count the number of records in `df[1]` containing `?`.

In [15]:
```
# CODE HERE - Task 3 Response
count = data[1].value_counts().get('?') #count number of ?
print(count)
```

8

Task 4

(1 PT)

Replace `'?'` with `np.nan` in the age column.

In [17]:
```
# CODE HERE - Replace ? with null values
data[1].replace('?', np.nan, inplace=True)
```

Task 5

(1 PT)

Print the number of records containing `np.nan` in the column `df[1]` of your dataframe.

In [19]:
```
# CODE HERE - count np.nan
count_nan = data[1].isna().sum() #
print(count_nan)
```

8

Another complication

Another complication: the age data is saved as strings, and there are the null values.

Here's an example:

```
# inspect first element
df[1].iloc[0]

'77'
# check the column type
df[1].dtype
```

```
dtype('O')
```

To convert the column to numeric, we can use `apply()` with a lambda function.

If the type is string, we cast to numeric, e.g. `float` or `int`, otherwise it's null and we leave things alone.

`isinstance(x, str)` checks if `x` is a string, returning a bool.

Review this code for understanding:

```
df[1] = df[1].apply(lambda x: float(x) if isinstance(x, str) else
x)
```

Task 6

(1 PT)

Run the lambda function above, then show the data type of `age` is no longer string type.

In [22]:
```
# CODE HERE - Convert column 1 data type to float
data[1] = data[1].apply(lambda x: float(x) if isinstance(x, str) else x)
data[1].dtype
```

Out[22]:
```
dtype('float64')
```

Task 7

(1 PT)

Compute the median age.

In [24]:
```
# CODE HERE - 7.compute median age
median = data[1].median()
print(median)
```

```
63.0
```

Working with GENDER

The third column contains patient gender.

Again, since indexing starts at zero, you'll reference `df[2]`.

Task 8

(1 PT)

Print the frequency AND percentage of each gender.

Hint: The function you'll use to compute frequencies will take an argument to compute normalized values, which may be converted to percentages.

In [27]:
```
# CODE HERE -
count_male = data[2].value_counts().get(1)
count_female = data[2].value_counts().get(0)
print(f'Num Males: {count_male}\t  Num Females: {count_female}')
print(f'Percentage males: {round(100*count_male/(count_male+count_female), 3
```

```
Num Males: 1065   Num Females: 635
Percentage males: 62.647%        Percentage Females: 37.353%
```

Working with Essential Hypertension (EH)

Reference this column with `df[8]`.

Task 9

(1 PT)

Enter the most frequent value.

In [30]:
```
# CODE HERE - Most common essential Hypertension value
most_freq_count = 0
for i in range(5):
    current = data[8].value_counts().get(i)
    if current > most_freq_count:
        most_freq_count = current
        most_freq_val = i

print(f'Most freq EH value: { most_freq_val}')
print(f'Most freq count: {most_freq_count} \n')

# Check Work using DDF as reference
print(data[8].value_counts().get(0))
print(data[8].value_counts().get(1))
print(data[8].value_counts().get(2))
print(data[8].value_counts().get(3))
print(data[8].value_counts().get(4))
```

```
Most freq EH value: 0
Most freq count: 880

880
605
195
11
9
```

```
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:4: FutureWarning: Series.__getitem__ treating keys as positions is deprec
ated. In a future version, integer keys will always be treated as labels (co
nsistent with DataFrame behavior). To access a value by position, use `ser.i
loc[pos]`
  current = data[8].value_counts().get(i)
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:4: FutureWarning: Series.__getitem__ treating keys as positions is deprec
ated. In a future version, integer keys will always be treated as labels (co
nsistent with DataFrame behavior). To access a value by position, use `ser.i
loc[pos]`
  current = data[8].value_counts().get(i)
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:4: FutureWarning: Series.__getitem__ treating keys as positions is deprec
ated. In a future version, integer keys will always be treated as labels (co
nsistent with DataFrame behavior). To access a value by position, use `ser.i
loc[pos]`
  current = data[8].value_counts().get(i)
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:4: FutureWarning: Series.__getitem__ treating keys as positions is deprec
ated. In a future version, integer keys will always be treated as labels (co
nsistent with DataFrame behavior). To access a value by position, use `ser.i
loc[pos]`
  current = data[8].value_counts().get(i)
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:4: FutureWarning: Series.__getitem__ treating keys as positions is deprec
ated. In a future version, integer keys will always be treated as labels (co
nsistent with DataFrame behavior). To access a value by position, use `ser.i
loc[pos]`
  current = data[8].value_counts().get(i)
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:13: FutureWarning: Series.__getitem__ treating keys as positions is depre
cated. In a future version, integer keys will always be treated as labels (c
onsistent with DataFrame behavior). To access a value by position, use `ser.
iloc[pos]`
  print(data[8].value_counts().get(0))
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:14: FutureWarning: Series.__getitem__ treating keys as positions is depre
cated. In a future version, integer keys will always be treated as labels (c
onsistent with DataFrame behavior). To access a value by position, use `ser.
iloc[pos]`
  print(data[8].value_counts().get(1))
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:15: FutureWarning: Series.__getitem__ treating keys as positions is depre
cated. In a future version, integer keys will always be treated as labels (c
onsistent with DataFrame behavior). To access a value by position, use `ser.
iloc[pos]`
  print(data[8].value_counts().get(2))
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:16: FutureWarning: Series.__getitem__ treating keys as positions is depre
cated. In a future version, integer keys will always be treated as labels (c
onsistent with DataFrame behavior). To access a value by position, use `ser.
iloc[pos]`
  print(data[8].value_counts().get(3))
/var/folders/95/w1686fhn34zczpnrnccjxvgh0000gn/T/ipykernel_62852/1986006245.
py:17: FutureWarning: Series.__getitem__ treating keys as positions is depre
```

```
cated. In a future version, integer keys will always be treated as labels (c
onsistent with DataFrame behavior). To access a value by position, use `ser.
iloc[pos]`
  print(data[8].value_counts().get(4))
```

Working with Atrial Fibrillation (AFIB)

Reference this column with `df[112]`.

AFIB is one of the complications and outcomes of myocardial infarction.

Task 10

(1 PT)

Print the number of AFIB cases.

Note that 1 means there is a case.

In [33]:
```
# CODE HERE - Print number of AFIB cases
number_AFIB = len(data[data[112]==1])
print(f'Number of AFIB cases: {number_AFIB}')
```

```
Number of AFIB cases: 170
```

Combining Age and AFIB

Task 11

(1 PT)

Construct a new dataframe `df2` containing only the columns for AGE and AFIB.

Recall that AGE is in `df[1]` and AFIB is in `df[112]`.

Print the shape of this dataframe.

Hint: you can pass a list of column names to the dataframe indexer to get a dataframe with a subset of columns.

In [36]:
```
# CODE HERE - new DataFrame from old one
df2 = data.iloc[:, [1,122]] # Use .iloc to select columns by index position
print(df2.head)
print(f'\n\nShape: {df2.shape}')
```

```
<bound method NDFrame.head of         1    122
0     77.0    0
1     55.0    0
2     52.0    0
3     68.0    0
4     60.0    0
...    ...  ...
1695  77.0    0
1696  70.0    0
1697  55.0    0
1698  79.0    0
1699  63.0    0

[1700 rows x 2 columns]>


Shape: (1700, 2)
```

Plotting

We are going to plot AGE and AFIB, so renaming the columns to strings will make our visualization more readable.

You can rename columns using the dataframe `.rename()` method, which takes a dictionary as an argument of the form:

```
{
    current_column_name1: new_column_name1,
    ...
    current_column_nameN: new_column_nameN
}
```

Rename column `1` to `'age'` and `2` to `'AFIB'` for `df2`.

In [44]:

```
# CODE HERE - Rename Columns
df2 = df2.rename(columns = {
        1: 'age',
        122: 'AFIB'
})

df2.head()
```

Out[44]:

	age	AFIB
0	77.0	0
1	55.0	0
2	52.0	0
3	68.0	0
4	60.0	0

Task 12

(1 PT)

Dispplay a boxplot with AFIB on the x-axis and Age on the y-axis

Use the the `boxplot()` function from the `seaborn` package for this.

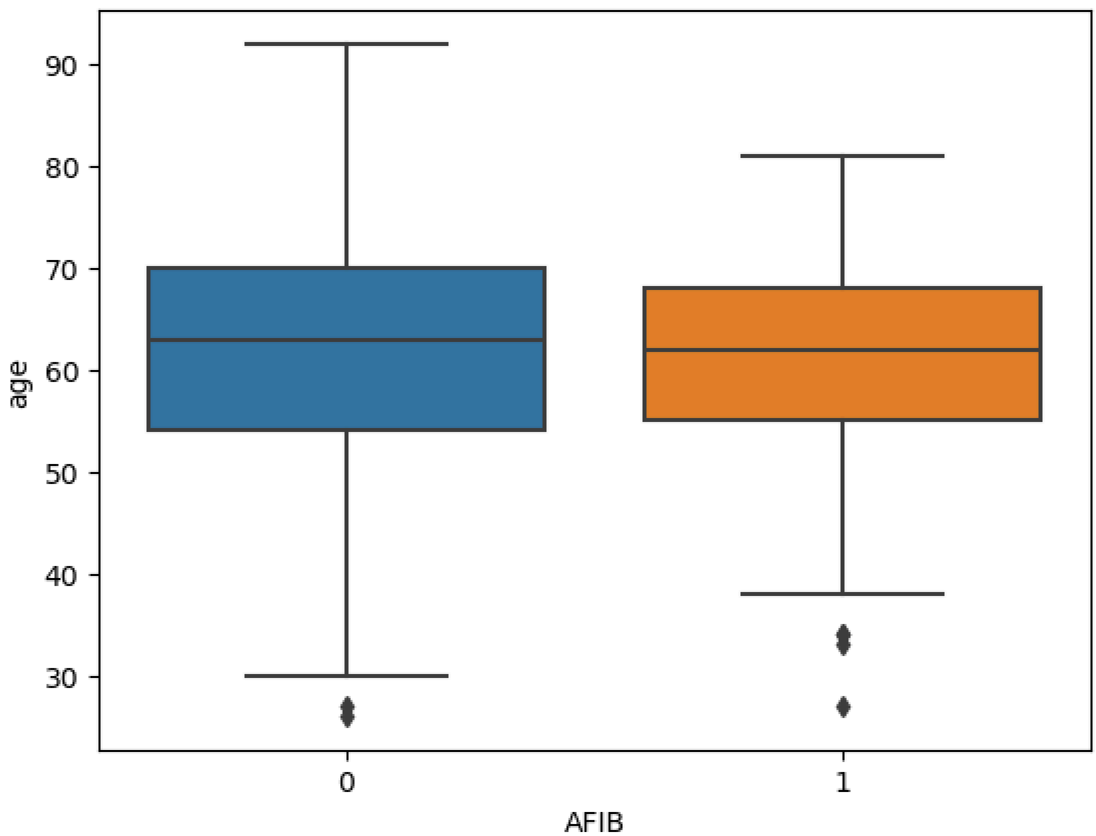
Here is the documentation, but all you need to do is this:

```
from seaborn import boxplot
```

In [46]:

```
# CODE HERE - Plotting AGE and AFIB
from seaborn import boxplot
boxplot(data=df2, x='AFIB', y='age')
```

Out[46]: <Axes: xlabel='AFIB', ylabel='age'>



Ungraded question: What do you notice about the difference in age distributions between AFIB and non-AFIB groups?

- Very Few people who have AFIB (1) were very young or very old -> this leads me to believe kids dont get it, and healthy people who live long lives also dont get it very frequently

In [ ]:

```

```