**Biniam Abebe - 04/20/2024**

# Hands-on Assignment

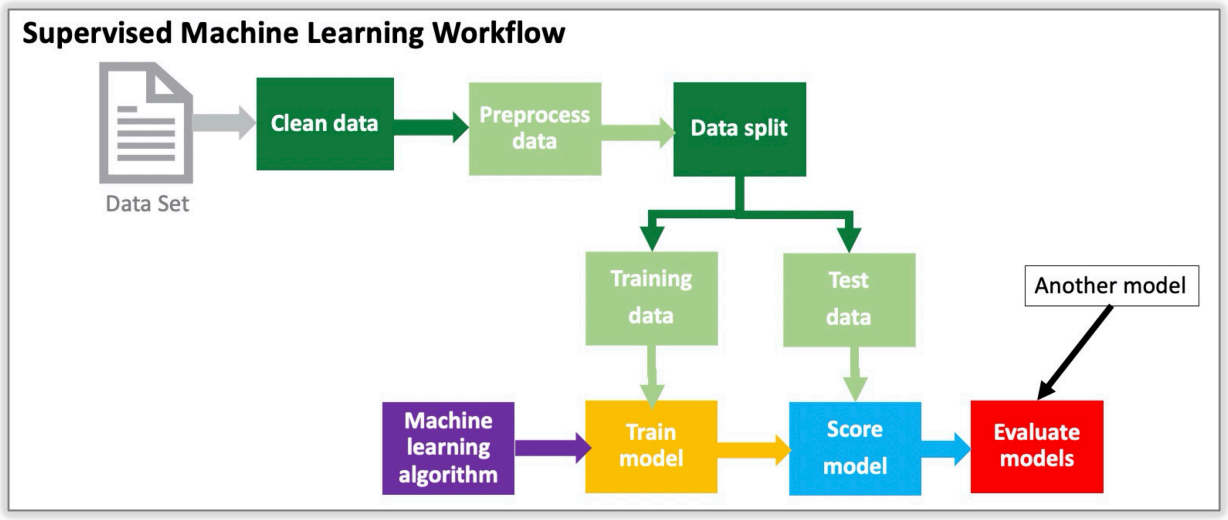
## Complete the following two sections on Supervised Machine Learning:

- Linear Regression
- Logistic Regression

# Linear and Logistic Regression

# Part 1: Linear Regression

## Machine Learning Supervised Linear Regression




- You can see in the workflow; supervised learning starts with the data set. Remember, since it is supervised, the data is labeled. Then there is some data preprocessing (cleaning) to be done. Next, you will declare your input (X/Independent variables) and output (Target Variable/Dependent or Y) NumPy Arrays. Then the data is split into a testing and training set. Then you will build and train the model, use the model for predictions, and evaluate/validate the model. So let's begin.!

## STEP 1: Import Libraries

- import pandas and numpy libraries
- import scatter_matrix from pandas.plotting

- import LinearRegression from sklearn.linear_model
- import train_test_split, KFold, and cross_val_score from sklearn.model_selection
- import matplotlib
- import seaborn

In [ ]:

```
#Add your code here
#Import Libraries
import pandas as pd
from pandas.plotting import scatter_matrix
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split, KFold, cross_val_score
import matplotlib.pyplot as plt
import seaborn as sns

#additonal need Library
import numpy as np
import os
# filter warnings
import warnings
warnings.filterwarnings("ignore")
```

# WORKFLOW: DATA SET

# STEP 2: Read data description and Load the Data

- Read the description of the dataset listed below
- Dataset is provided in the module and assignment. It is called housing_boston.csv.
- Load the data into Pandas dataframe called df
- View the first five rows of the dataframe

## Description of Boston Housing Dataset

- CRIM: This is the per capita crime rate by town
- ZN: This is the proportion of residential land zoned for lots larger than 25,000 sq. ft.
- INDUS: This is the proportion of non-retail business acres per town.
- CHAS: This is the Charles River dummy variable (this is equal to 1 if tract bounds river; 0 otherwise)
- NOX: This is the concentration of the nitric oxide (parts per 10 million)
- RM: This is the average number of rooms per dwelling
- AGE: This is the proportion of owner-occupied units built prior to 1940
- DIS: This is the weighted distances to five Boston employment centers
- RAD: This is the index of accessibility to radial highways
- TAX: This is the full-value property-tax rate per 10,000 dollars
- PTRATIO: This is the pupil-teacher ratio by town
- AA: This is calculated as $1000(AA — 0.63)^2$, where AA is the proportion of people of African American descent by town
- LSTAT: This is the percentage lower status of the population

- MEDV: This is the median value of owner-occupied homes in $1000s

In [ ]:
```
#Add your code here
# Specify the location of the dataset.
housingfile = 'housing_boston.csv'
```

In [ ]:
```
#Add your code here
# Load the data into a Pandas DataFrame
df= pd.read_csv (housingfile, header=None)
```

In [ ]:
```
# Display the first 5 rows of the dataset
df.head()
```

Out[ ]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | 0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1 | 296 | 15.3 | 396.90 | 4.98 | 24.0 |
| **1** | 0.02731 | 0.0 | 7.07 | 0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2 | 242 | 17.8 | 396.90 | 9.14 | 21.6 |
| **2** | 0.02729 | 0.0 | 7.07 | 0 | 0.469 | 7.185 | 61.1 | 4.9671 | 2 | 242 | 17.8 | 392.83 | 4.03 | 34.7 |
| **3** | 0.03237 | 0.0 | 2.18 | 0 | 0.458 | 6.998 | 45.8 | 6.0622 | 3 | 222 | 18.7 | 394.63 | 2.94 | 33.4 |
| **4** | 0.06905 | 0.0 | 2.18 | 0 | 0.458 | 7.147 | 54.2 | 6.0622 | 3 | 222 | 18.7 | 396.90 | 5.33 | 36.2 |

# STEP 3: Give names to the columns since there are no headers

- Give the following names to the columns: 'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX','RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'AA', 'LSTAT', 'MEDV'
- Verify columns names were added
- View the first five rows of the dataframe

In [ ]:
```
#Add your code here
#give names to the columns columns:'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX','RM', 'AGE', 'DIS
col_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX','RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRA
```

In [ ]:
```
#Add your code here
# Let's check to see if the column names were added
df.columns = col_names
```

In [ ]:
```
# Look at the first 5 rows of data
df.head()
```

Out[ ]:

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | AA | LSTAT | MEDV |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | 0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1 | 296 | 15.3 | 396.90 | 4.98 | 24.0 |
| **1** | 0.02731 | 0.0 | 7.07 | 0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2 | 242 | 17.8 | 396.90 | 9.14 | 21.6 |

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | AA | LSTAT | MEDV |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2** | 0.02729 | 0.0 | 7.07 | 0 | 0.469 | 7.185 | 61.1 | 4.9671 | 2 | 242 | 17.8 | 392.83 | 4.03 | 34.7 |
| **3** | 0.03237 | 0.0 | 2.18 | 0 | 0.458 | 6.998 | 45.8 | 6.0622 | 3 | 222 | 18.7 | 394.63 | 2.94 | 33.4 |
| **4** | 0.06905 | 0.0 | 2.18 | 0 | 0.458 | 7.147 | 54.2 | 6.0622 | 3 | 222 | 18.7 | 396.90 | 5.33 | 36.2 |

# WORKFLOW: Clean and Preprocess the Dataset

# STEP 4: Clean the data

- Find and Mark Missing Values
- If there are no missing data points, then proceed to Step 5.

In [ ]:
```
#Add your code here
df.isnull().sum()
# We see there are no missing data points
```

Out[ ]:
```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
RM         0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
AA         0
LSTAT      0
MEDV       0
dtype: int64
```

# STEP 5: Performing the Exploratory Data Analysis (EDA)

- Print a count of the number of rows (observations) and columns (variables)
- Print the data types of all variables
- Print a summary statistics of the data

In [ ]:
```
# Get the number of records/rows and the number of variables/column
#Add your code here
print(df.shape)
```

```
(452, 14)
```

In [ ]:
```
# Get the data types of all variables
#Add your code here
print(df.dtypes)
```

```
CRIM       float64
ZN         float64
INDUS      float64
```

```
CHAS        int64
NOX       float64
RM        float64
AGE       float64
DIS       float64
RAD         int64
TAX         int64
PTRATIO   float64
AA        float64
LSTAT     float64
MEDV      float64
dtype: object
```

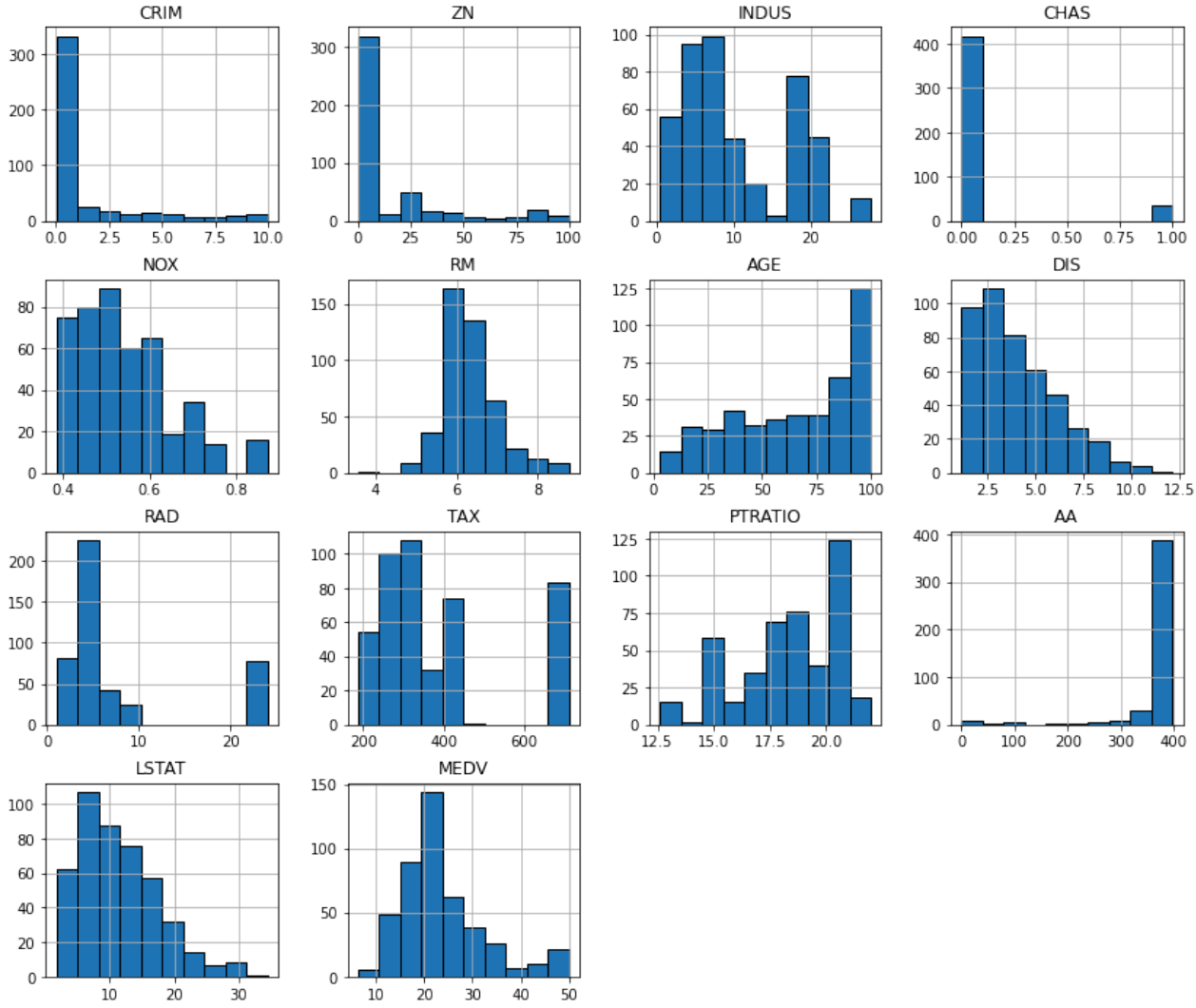
In [ ]:
```
# Obtain the summary statistics of the data
#Add your code here
print(df.describe())
```

```
             CRIM          ZN       INDUS        CHAS         NOX          RM  \
count  452.000000  452.000000  452.000000  452.000000  452.000000  452.000000
mean     1.420825   12.721239   10.304889    0.077434    0.540816    6.343538
std      2.495894   24.326032    6.797103    0.267574    0.113816    0.666808
min      0.006320    0.000000    0.460000    0.000000    0.385000    3.561000
25%      0.069875    0.000000    4.930000    0.000000    0.447000    5.926750
50%      0.191030    0.000000    8.140000    0.000000    0.519000    6.229000
75%      1.211460   20.000000   18.100000    0.000000    0.605000    6.635000
max      9.966540  100.000000   27.740000    1.000000    0.871000    8.780000

              AGE         DIS         RAD         TAX     PTRATIO          AA  \
count  452.000000  452.000000  452.000000  452.000000  452.000000  452.000000
mean    65.557965    4.043570    7.823009  377.442478   18.247124  369.826504
std     28.127025    2.090492    7.543494  151.327573    2.200064   68.554439
min      2.900000    1.129600    1.000000  187.000000   12.600000    0.320000
25%     40.950000    2.354750    4.000000  276.750000   16.800000  377.717500
50%     71.800000    3.550400    5.000000  307.000000   18.600000  392.080000
75%     91.625000    5.401100    7.000000  411.000000   20.200000  396.157500
max    100.000000   12.126500   24.000000  711.000000   22.000000  396.900000

            LSTAT        MEDV
count  452.000000  452.000000
mean    11.441881   23.750442
std      6.156437    8.808602
min      1.730000    6.300000
25%      6.587500   18.500000
50%     10.250000   21.950000
75%     15.105000   26.600000
max     34.410000   50.000000
```

# STEP 5A: Create Histograms

- Create histograms for each variable from the dataframe df with a figure size of 14 x 12
- Plot the histograms

In [ ]:
```
# Plot histogram for each variable. I encourage you to work with the histogram.
# Create histograms for each variable from the dataframe df with a figure size of 14 x 12
# Plot the histograms
#Add your code here
df.hist(edgecolor= 'black',figsize=(14,12))
plt.show()
```

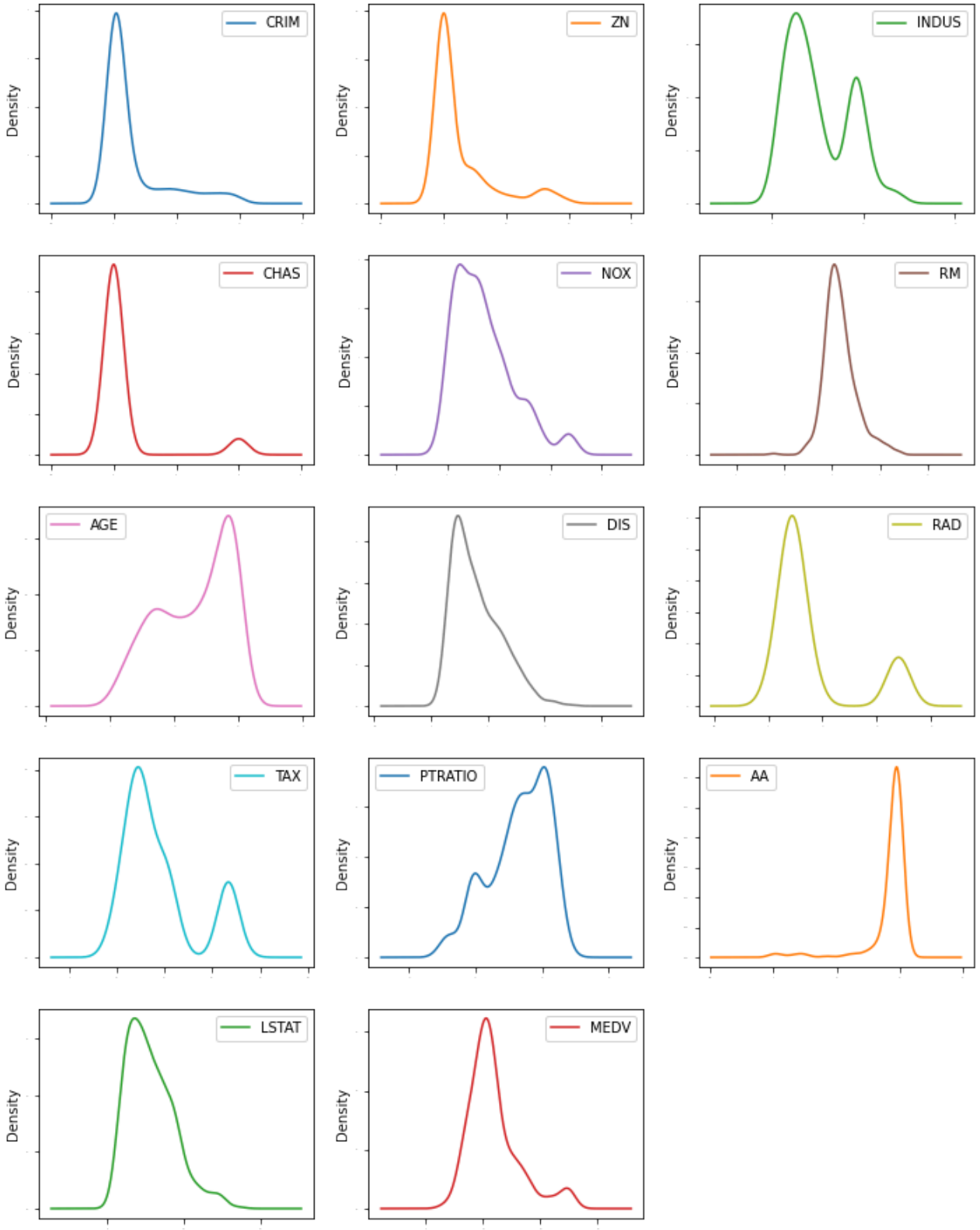



# STEP 5B: Create Density Plots

- Create density plots from the dataframe df that 14 numeric variables, at least 14 plots, layout (5,3): 5 rows, each row with 3 plots
- Plot the density plots

In [ ]:

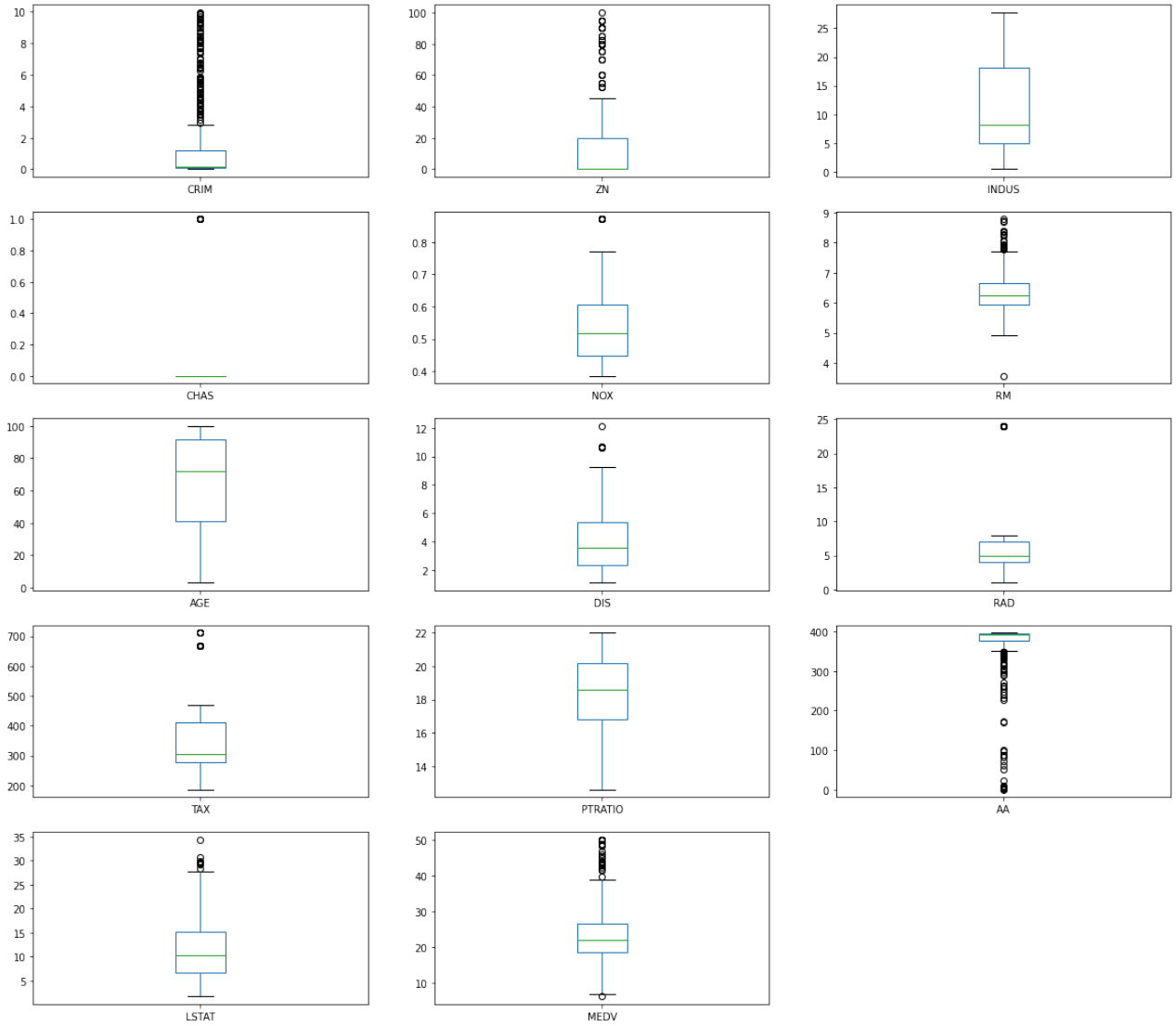
```
# Density plots
# Notes: 14 numeric variables, at least 14 plots, layout (5,3): 5 rows, each row with 3 p
#Add your code here
df.plot(kind='density', subplots=True, layout= (5,3), sharex=False, legend=True, fontsize
plt.show()
```




# STEP 5C: Create Boxplots

- Create a boxplots from the dataframe df with a layout (5,3) and figure size (20,18). Ensure subplots is True and sharex is False.
- Plot the boxplots

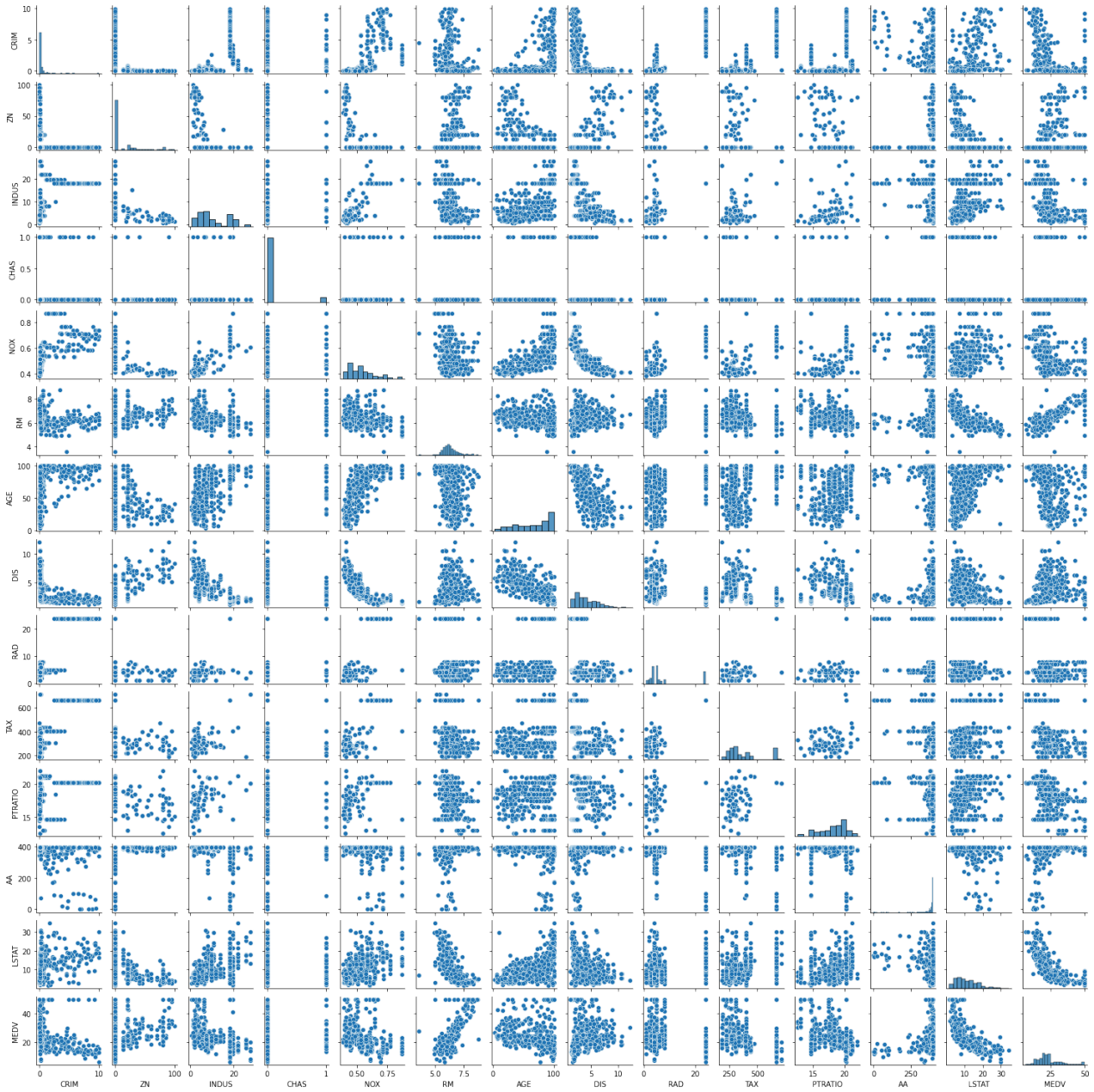
In [ ]:
```
# Create a boxplots from the dataframe df with a layout (5,3) and figure size (20,18). En
# Plot the boxplots
#Add your code here
df.plot(kind="box", subplots=True, layout=(5,3), sharex=False, figsize=(20,18))
plt.show()
```




# STEP 5D: Pair Plots - Correlation Analysis and Feature Selection

- Create pair plots of the dataframe with a height of 1.5
- Plot the pair plots
- Use the format function to decrease the number of decimal places to three
- Obtain the correlation of the dataframe

In [ ]:
```
#Obtain pair plots of the data. I know this is a lot of information, but I wanted you to
#Add your code here
sns.pairplot(df, height=1.5);
plt.show()
```




In [ ]:
```
# We will decrease the number of decimal places with the format function.
#Add your code here
pd.options.display.float_format = '{:,.3f}'.format
```

In [ ]:
```
# Here we will get the correlations, with only 3 decimals.
#Add your code here
df.corr()
```

Out[ ]:

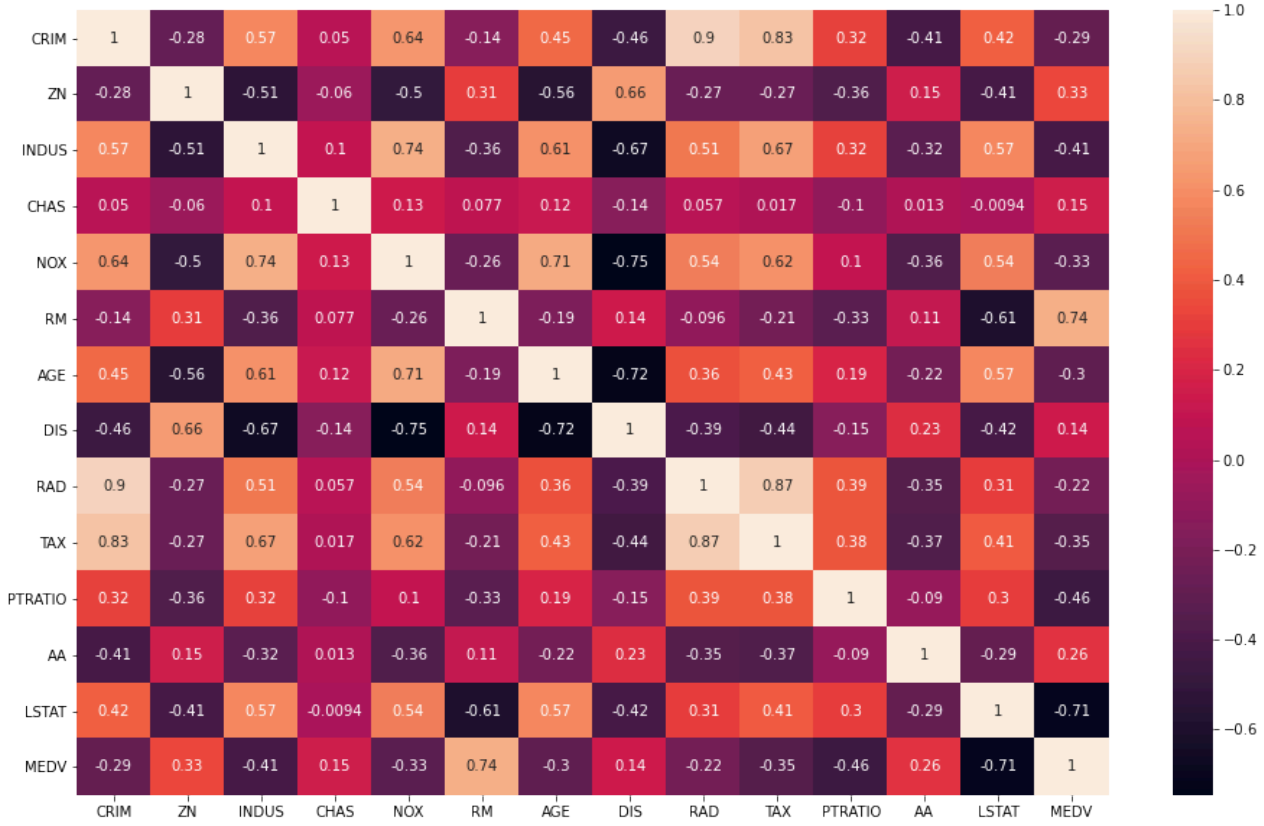
| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | AA | LST |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **CRIM** | 1.000 | -0.281 | 0.574 | 0.050 | 0.637 | -0.142 | 0.448 | -0.462 | 0.898 | 0.826 | 0.319 | -0.413 | 0.4 |
| **ZN** | -0.281 | 1.000 | -0.514 | -0.060 | -0.501 | 0.307 | -0.556 | 0.656 | -0.267 | -0.269 | -0.364 | 0.150 | -0.4 |
| **INDUS** | 0.574 | -0.514 | 1.000 | 0.103 | 0.739 | -0.365 | 0.606 | -0.669 | 0.513 | 0.673 | 0.317 | -0.317 | 0.5 |
| **CHAS** | 0.050 | -0.060 | 0.103 | 1.000 | 0.134 | 0.077 | 0.123 | -0.141 | 0.057 | 0.017 | -0.100 | 0.013 | -0.0 |
| **NOX** | 0.637 | -0.501 | 0.739 | 0.134 | 1.000 | -0.265 | 0.707 | -0.746 | 0.542 | 0.615 | 0.103 | -0.358 | 0.5 |

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | AA | LST |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **RM** | -0.142 | 0.307 | -0.365 | 0.077 | -0.265 | 1.000 | -0.188 | 0.139 | -0.096 | -0.215 | -0.334 | 0.108 | -0.( |
| **AGE** | 0.448 | -0.556 | 0.606 | 0.123 | 0.707 | -0.188 | 1.000 | -0.720 | 0.359 | 0.427 | 0.193 | -0.224 | 0.! |
| **DIS** | -0.462 | 0.656 | -0.669 | -0.141 | -0.746 | 0.139 | -0.720 | 1.000 | -0.388 | -0.444 | -0.152 | 0.234 | -0.4 |
| **RAD** | 0.898 | -0.267 | 0.513 | 0.057 | 0.542 | -0.096 | 0.359 | -0.388 | 1.000 | 0.873 | 0.387 | -0.353 | 0.: |
| **TAX** | 0.826 | -0.269 | 0.673 | 0.017 | 0.615 | -0.215 | 0.427 | -0.444 | 0.873 | 1.000 | 0.385 | -0.367 | 0.4 |
| **PTRATIO** | 0.319 | -0.364 | 0.317 | -0.100 | 0.103 | -0.334 | 0.193 | -0.152 | 0.387 | 0.385 | 1.000 | -0.090 | 0.: |
| **AA** | -0.413 | 0.150 | -0.317 | 0.013 | -0.358 | 0.108 | -0.224 | 0.234 | -0.353 | -0.367 | -0.090 | 1.000 | -0.: |
| **LSTAT** | 0.425 | -0.411 | 0.565 | -0.009 | 0.537 | -0.607 | 0.573 | -0.424 | 0.310 | 0.411 | 0.303 | -0.291 | 1.( |
| **MEDV** | -0.286 | 0.332 | -0.412 | 0.154 | -0.333 | 0.740 | -0.300 | 0.139 | -0.218 | -0.346 | -0.461 | 0.265 | -0.: |

# STEP 5E: Creating Heatmaps

- Create a heatmap of the dataframe with a figure size of 16x10, use the dataframe correlation and ensure annot is True
- Plot the heatmap
- For help type sns.heatmap?
- Create another dataframe df2 with less variables
- Obtain the correlation of the dataframe df2
- Plot pair plot for df2 with a height of 5.5
- Create a heatmap of the dataframe df2 with a figure size of 20x12, use the dataframe correlation and ensure annot is True
- Change the color to blue and increase the font size 20

In [ ]:
```
# We could simply look at the correlations, but a heatmap is a great way to present to th
#Add your code here
plt.figure(figsize=(16,10))
sns.heatmap(df.corr(), annot=True)
plt.show()
```




In [ ]:
```
# If you get stuck on what can be done with the heatmap, you can use the following code t
#Add your code here
help(sns.heatmap)
```

```
Help on function heatmap in module seaborn.matrix:

heatmap(data, *, vmin=None, vmax=None, cmap=None, center=None, robust=False, annot=None, f
mt='.2g', annot_kws=None, linewidths=0, linecolor='white', cbar=True, cbar_kws=None, cbar_
ax=None, square=False, xticklabels='auto', yticklabels='auto', mask=None, ax=None, **kwarg
s)
    Plot rectangular data as a color-encoded matrix.

    This is an Axes-level function and will draw the heatmap into the
    currently-active Axes if none is provided to the ``ax`` argument.  Part of
    this Axes space will be taken and used to plot a colormap, unless ``cbar``
    is False or a separate Axes is provided to ``cbar_ax``.

    Parameters
    ----------
    data : rectangular dataset
        2D dataset that can be coerced into an ndarray. If a Pandas DataFrame
        is provided, the index/column information will be used to label the
        columns and rows.
    vmin, vmax : floats, optional
        Values to anchor the colormap, otherwise they are inferred from the
        data and other keyword arguments.
    cmap : matplotlib colormap name or object, or list of colors, optional
        The mapping from data values to color space. If not provided, the
        default will depend on whether ``center`` is set.
    center : float, optional
        The value at which to center the colormap when plotting divergant data.
        Using this parameter will change the default ``cmap`` if none is
        specified.
    robust : bool, optional
        If True and ``vmin`` or ``vmax`` are absent, the colormap range is
```

```
    computed with robust quantiles instead of the extreme values.
annot : bool or rectangular dataset, optional
    If True, write the data value in each cell. If an array-like with the
    same shape as ``data``, then use this to annotate the heatmap instead
    of the data. Note that DataFrames will match on position, not index.
fmt : str, optional
    String formatting code to use when adding annotations.
annot_kws : dict of key, value mappings, optional
    Keyword arguments for :meth:`matplotlib.axes.Axes.text` when ``annot``
    is True.
linewidths : float, optional
    Width of the lines that will divide each cell.
linecolor : color, optional
    Color of the lines that will divide each cell.
cbar : bool, optional
    Whether to draw a colorbar.
cbar_kws : dict of key, value mappings, optional
    Keyword arguments for :meth:`matplotlib.figure.Figure.colorbar`.
cbar_ax : matplotlib Axes, optional
    Axes in which to draw the colorbar, otherwise take space from the
    main Axes.
square : bool, optional
    If True, set the Axes aspect to "equal" so each cell will be
    square-shaped.
xticklabels, yticklabels : "auto", bool, list-like, or int, optional
    If True, plot the column names of the dataframe. If False, don't plot
    the column names. If list-like, plot these alternate labels as the
    xticklabels. If an integer, use the column names but plot only every
    n label. If "auto", try to densely plot non-overlapping labels.
mask : bool array or DataFrame, optional
    If passed, data will not be shown in cells where ``mask`` is True.
    Cells with missing values are automatically masked.
ax : matplotlib Axes, optional
    Axes in which to draw the plot, otherwise use the currently-active
    Axes.
kwargs : other keyword arguments
    All other keyword arguments are passed to
    :meth:`matplotlib.axes.Axes.pcolormesh`.

Returns
-------
ax : matplotlib Axes
    Axes object with the heatmap.

See Also
--------
clustermap : Plot a matrix using hierachical clustering to arrange the
             rows and columns.

Examples
--------

Plot a heatmap for a numpy array:

.. plot::
    :context: close-figs

    >>> import numpy as np; np.random.seed(0)
    >>> import seaborn as sns; sns.set_theme()
    >>> uniform_data = np.random.rand(10, 12)
    >>> ax = sns.heatmap(uniform_data)

Change the limits of the colormap:

.. plot::
```

```
    :context: close-figs

    >>> ax = sns.heatmap(uniform_data, vmin=0, vmax=1)

Plot a heatmap for data centered on 0 with a diverging colormap:

.. plot::
    :context: close-figs

    >>> normal_data = np.random.randn(10, 12)
    >>> ax = sns.heatmap(normal_data, center=0)

Plot a dataframe with meaningful row and column labels:

.. plot::
    :context: close-figs

    >>> flights = sns.load_dataset("flights")
    >>> flights = flights.pivot("month", "year", "passengers")
    >>> ax = sns.heatmap(flights)

Annotate each cell with the numeric value using integer formatting:

.. plot::
    :context: close-figs

    >>> ax = sns.heatmap(flights, annot=True, fmt="d")

Add lines between each cell:

.. plot::
    :context: close-figs

    >>> ax = sns.heatmap(flights, linewidths=.5)

Use a different colormap:

.. plot::
    :context: close-figs

    >>> ax = sns.heatmap(flights, cmap="YlGnBu")

Center the colormap at a specific value:

.. plot::
    :context: close-figs

    >>> ax = sns.heatmap(flights, center=flights.loc["Jan", 1955])

Plot every other column label and don't plot row labels:

.. plot::
    :context: close-figs

    >>> data = np.random.randn(50, 20)
    >>> ax = sns.heatmap(data, xticklabels=2, yticklabels=False)

Don't draw a colorbar:

.. plot::
    :context: close-figs

    >>> ax = sns.heatmap(flights, cbar=False)

Use different axes for the colorbar:
```

```
    .. plot::
        :context: close-figs

        >>> grid_kws = {"height_ratios": (.9, .05), "hspace": .3}
        >>> f, (ax, cbar_ax) = plt.subplots(2, gridspec_kw=grid_kws)
        >>> ax = sns.heatmap(flights, ax=ax,
        ...                  cbar_ax=cbar_ax,
        ...                  cbar_kws={"orientation": "horizontal"})

    Use a mask to plot only part of a matrix

    .. plot::
        :context: close-figs

        >>> corr = np.corrcoef(np.random.randn(10, 200))
        >>> mask = np.zeros_like(corr)
        >>> mask[np.triu_indices_from(mask)] = True
        >>> with sns.axes_style("white"):
        ...     f, ax = plt.subplots(figsize=(7, 5))
        ...     ax = sns.heatmap(corr, mask=mask, vmax=.3, square=True)
```

In [ ]:

```
# Now, let's say we want to decrease the number of variables in our heatmap.
# Remember how to make a subset. Try using different variables.
#Add your code here
df2= df[['CRIM','INDUS', 'TAX','MEDV']]
```

In [ ]:

```
# Here we will look at the correlations for only the variables in df2.
#Add your code here
df2.corr()
```

Out[ ]:

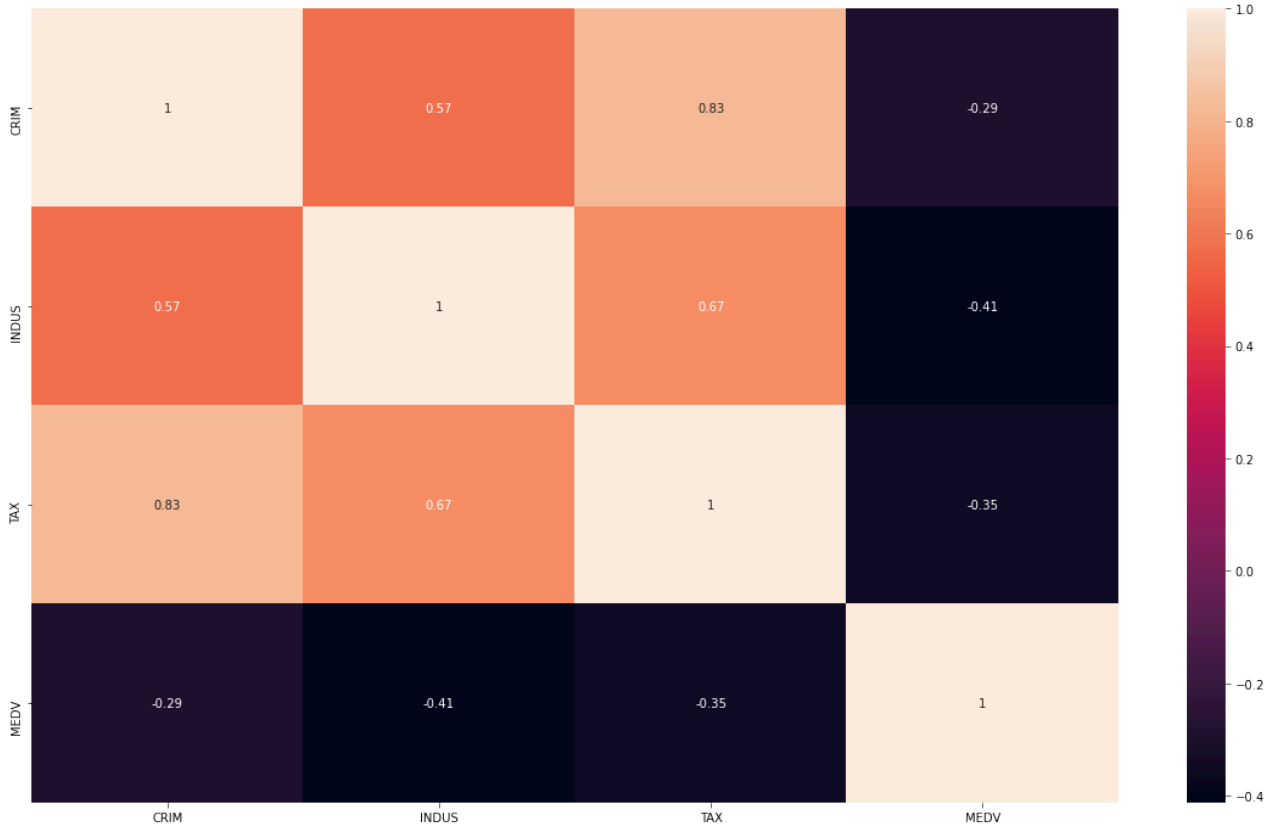
| | CRIM | INDUS | TAX | MEDV |
|---|---|---|---|---|
| **CRIM** | 1.000 | 0.574 | 0.826 | -0.286 |
| **INDUS** | 0.574 | 1.000 | 0.673 | -0.412 |
| **TAX** | 0.826 | 0.673 | 1.000 | -0.346 |
| **MEDV** | -0.286 | -0.412 | -0.346 | 1.000 |

In [ ]:

```
# Let's try the pairplot with only the variables in df2
# Plot pair plot for df2 with a height of 5.5
#Add your code here
sns.pairplot(df2, height=5.5)
plt.show()
```

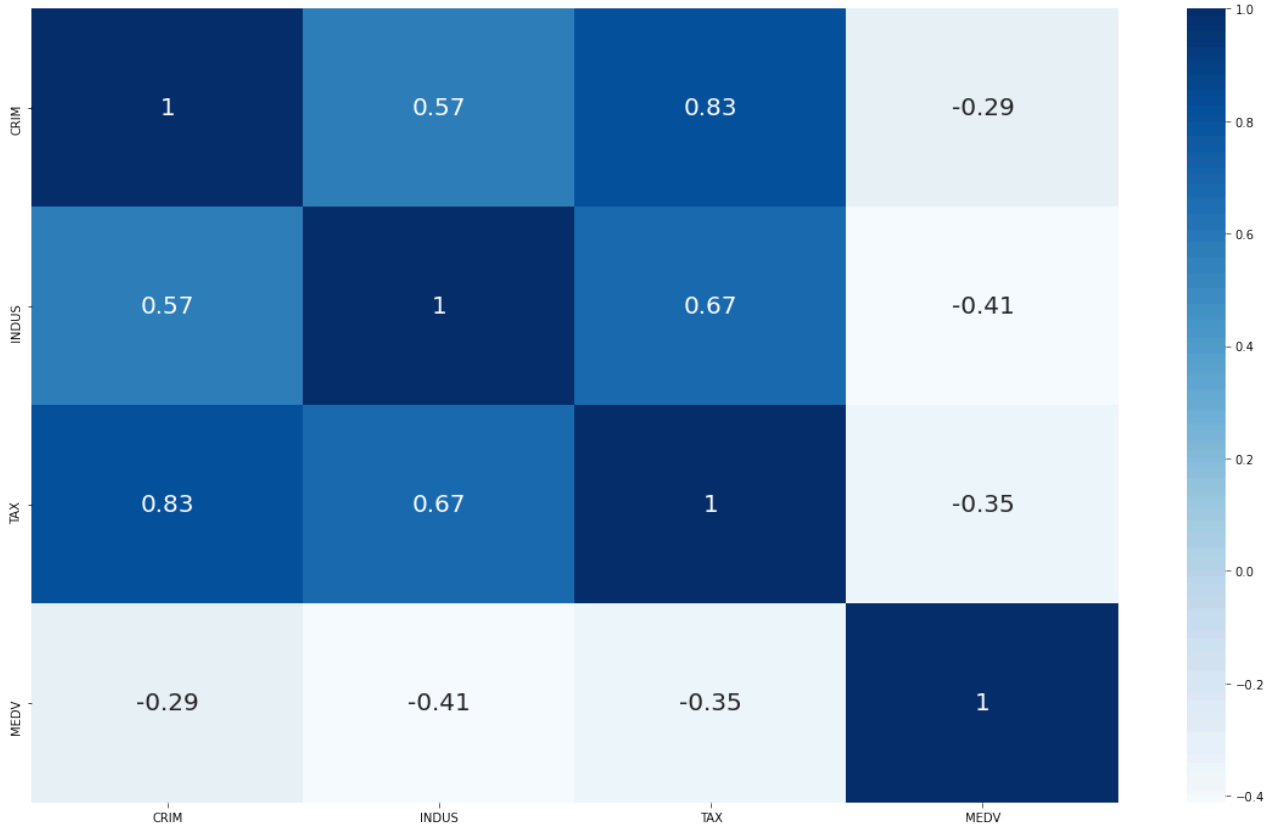

In [ ]:

```
# Now, we will make a heatmap with only the variables in the df2 subset.
# Create a heatmap of the dataframe df2 with a figure size of 20x12, use the dataframe co
# Change the color to blue and increase the font size 20
#Add your code here
plt.figure(figsize =(20,12))
sns.heatmap(df2.corr(), annot=True)
plt.show()
```




In [ ]:

```
#If you want to change the color and font to make the labels easier to read, use this cod
#Add your code here
plt.figure(figsize =(20,12))
sns.heatmap(df2.corr(), cmap="Blues", annot=True, annot_kws={"fontsize":20})
plt.show()
```



# WORKFLOW: DATA SPLIT

## STEP 6: Separate the Dataset into Input & Output NumPy Arrays

- Store the dataframe d2 values into a NumPy array
- Separate the array into input and output components by slicing

In [ ]:
```
# Store the dataframe values into a NumPy array
array= df2.values
# Separate the array into input and output components by slicing
# For X (input) [:, 0:13] --> all the rows, columns from 0 - 12 (13 - 1)
X = array[:,0:3]
# For Y (output) [:3] --> All the rows in the last column (MEDV)
Y = array [:,3]
```

## STEP 7: Split into Input/Output Array into Training/Testing Datasets

- Split the dataset into training at 67% and test at 33% with the seed = 7

In [ ]:
```
# Split the dataset --> training sub-dataset: 67%, and test sub-dataset:33%
test_size = 0.33
# Selection of records to include in which sub-dataset must be done randomly - use the fo
seed = 7
# Split the dataset (both input & output) into training/testing datasets
#Add your code here
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=test_size, random_sta
```

# WORKFLOW: TRAIN MODEL

## STEP 8: Build and Train the Model

- Assign LinearRegression to the model
- Train the model
- Print the intercept and coefficients
- Print the list of the coefficients with their correspondent variable name

In [ ]:
```
#Add your codes here
# Build the model
model = LinearRegression()
# Train the model using the training sub-dataset
model.fit(X_train, Y_train)
# Print intercept and coefficients
print("Intercept:",model.intercept_)
print("Coefficients:",model.coef_)
```

```
Intercept: 31.561375131572596
Coefficients: [-0.05240879 -0.45185299 -0.00915631]
```

In [ ]:
```
# Print out the list of the coefficients with their correspondent variable name
# Pair the feature names with the coefficients
names_ = df2.columns[0:3]
coefficients = model.coef_
names_coeff = list(zip(names_, coefficients))

# Convert iterator into a set
names_coeff_set = set(names_coeff)

#print
for coef in names_coeff_set:
    print (coef, "\n")
```

```
('TAX', -0.009156313349060512)

('INDUS', -0.45185299387878497)

('CRIM', -0.05240879307447724)
```

In [ ]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Out[ ]:
```
LinearRegression(n_jobs=1)
```

# WORKFLOW: SCORE MODEL

## STEP 9: Calculate R-Squared

- Calculate the R-Squared
- Print the score

** Note: The higher the R-squared, the better (0 – 100%). Depending on the model, the best models score above 83%. The R-squared value tells us how well the independent variables predict the dependent variable, which is very low. Think about how you could increase the R-squared. What variables would you use?

In [ ]:
```
#Add your code here
# Calculate the R-Squared
# Print the score
R_squared = model.score(X_test, Y_test)
print(R_squared)
```

```
0.07560379703404274
```

## Step 10: Prediction

- Execute model prediction
- We have now trained the model. Let's use the trained model to predict the "Median value of owner-occupied homes in 1000 dollars" (MEDV).

- We are using the following predictors:

- CRIM: per capita crime rate by town: 12
- INDUS: proportion of non-retail business acres per town: 10
- TAX: full-value property-tax rate per $10,000: 450

** Note: The model predicts that the median value of owner-occupied homes in 1000 dollars in the above suburb should be around $24,144.

In [ ]:
```
#Add your code here
# Execute model prediction
model.predict([[12,10,450]])
```

Out[ ]:
```
array([22.29359867])
```

# WORKFLOW: EVALUATE MODELS

## Step 11: Train & Score Model 2 Using K-Fold Cross Validation Data Split

- Specify the k-size to 10
- Fix the random seed to 7
- Split the entire data set
- Obtain the Mean squared error
- Train the model and run K-fold cross-validation
- Print results

In [ ]:
```
#Add your codes here

#Add Your Code Here# Evaluate the algorithm
# Specify the K-size
num_folds = 10
# Fix the random seed
# must use the same seed value so that the same subsets can be obtained
# for each time the process is repeated
seed = 7
# Split the whole data set into folds
kfold= KFold(n_splits=num_folds, random_state=seed, shuffle=True)

# For Linear regression, we can use MSE (mean squared error) value
# to evaluate the model/algorithm

scoring = 'neg_mean_squared_error'
```

In [ ]:
```
#Add your codes here

# Train the model and run K-foLd cross-validation to validate/evaluate the model
results = cross_val_score(model, X, Y, cv=kfold, scoring=scoring)

# Print out the evaluation results
```

```
# Result: the average of all the results obtained from the k-fold crossvalidation
print("Average of all results from the K-fold Cross-Validation, using negative mean squar
```

```
Average of all results from the K-fold Cross-Validation, using negative mean squared erro
r: -64.35862748210982
```

Note: After we train, we evaluate. We are using K-fold to determine if the model is acceptable. We pass the whole set since the system will divide it for us. We see a -64 avg of all errors (mean of square errors). This value would traditionally be positive, but scikit reports this value as a negative value. If the square root had been evaluated, the value would have been around 8.

# Step 12: Score Using Explained Variance

Let's use a different scoring parameter. Here we use the Explained Variance. The best possible score is 1.0; lower values are worse.

- Specify the k-size to 10
- Set the seed to 7
- Split the entire data set
- Obtain the explained variance score
- Train the model and run K-fold cross-validation
- Print results

In [ ]:

```
#Add your codes here

# Evaluate the algorithm
# Specify the K-size
num_folds = 10
# Fix the random seed must use the same seed value so that the same subsets can be obtain
# for each time the process is repeated
seed = 7
# Split the whole data set into folds
kfold= KFold(n_splits=num_folds, random_state=seed, shuffle=True)

# For Linear regression, we can use explained variance value to evaluate the model/algori
scoring = 'explained_variance'
```

In [ ]:

```
#Add your codes here

# Train the model and run K-foLd cross-validation to validate/evaluate the model
results = cross_val_score(model, X, Y, cv=kfold, scoring=scoring)

# Print out the evaluation results
# Result: the average of all the results obtained from the k-fold crossvalidation
print("Average of all results from the K-fold Cross Validation, using exlpained variance:
```

```
Average of all results from the K-fold Cross Validation, using exlpained variance: 0.19023
822025958698
```

To learn more about Scikit Learning scoring
[https://scikitlearn.org/stable/modules/model_evaluation.html (Links to an external site.)]

# Part 2: Logistic Regression

## Machine Learning Supervised Logistic Regression

• Let's begin Part 2 using logistic regression using the same Supervised Learning Workflow used in part 1.

## STEP 1: Import Libraries

- import pandas and numpy libraries
- import scatter_matrix from pandas.plotting
- import matplotlib
- import seaborn
- import LogisticRegression from sklearn.linear_model
- import train_test_split, KFold, and cross_val_score from sklearn.model_selection
- import classification_report from sklearn.metrics

In [ ]:
```
# Import Python Libraries: NumPy and Pandas
import pandas as pd
import numpy as np
```

In [ ]:
```
#Add Your Code Here# Import Libraries & modules for data visualization
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

In [ ]:
```
# Import scikit-Learn module for the algorithm/modeL: Logistic Regression
from sklearn.linear_model import LogisticRegression
```

In [ ]:
```
# Import scikit-Learn module to split the dataset into train/ test subdatasets
from sklearn.model_selection import train_test_split
```

In [ ]:
```
# Import scikit-Learn module for K-fold cross-validation - algorithm/modeL evaluation & v
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
```

In [ ]:
```
# Import scikit-Learn module classification report to later use for information about how
from sklearn.metrics import classification_report
```

## WORKFLOW: DATA SET

## STEP 2: Read data description and Load the Data

- Read the description of the dataset listed below
- Dataset is provided in the module and assignment. It is called iris.csv.
- Load the data into Pandas dataframe called df
- View the first five rows of the dataframe

# Description Iris Dataset

Data Set: iris.csv

Title: Iris Plants Database Updated Sept 21 by C. Blake -Added discrepancy information Sources:

- Creator: RA_ Fisher
- Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
- Date: 1988

Relevant Information: This is perhaps the best-known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field and is referenced frequently to this day. (See Duda & Hart, for example)

The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant.

Predicted attribute: class of Iris plant

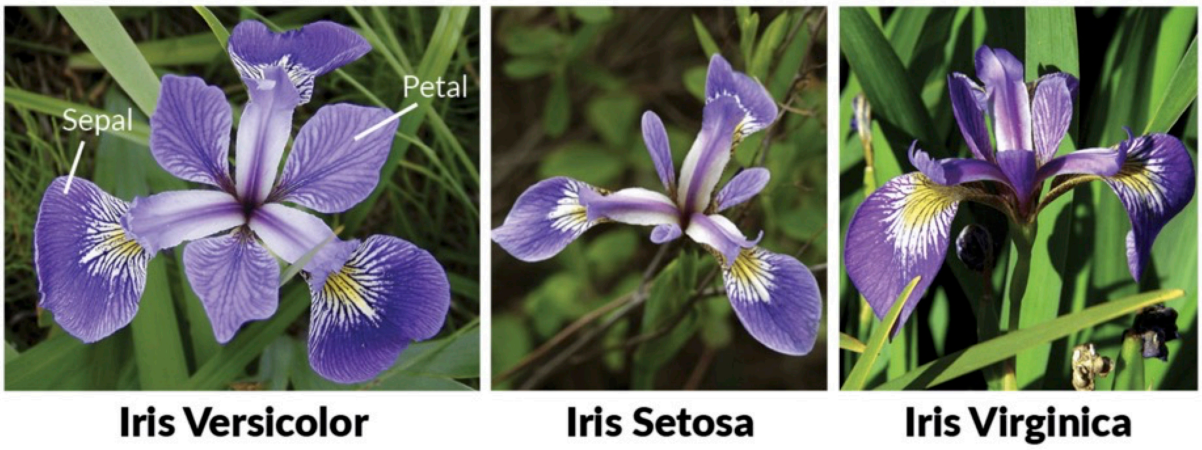
Number of Instances: 150 (50 in each of three classes)

Number of predictors: 4 numeric

Predictive attributes and the class attribute information:

- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm

class:



In [ ]:
```
# Specify the location of the dataset
filename = 'iris.csv'
# Load the data into a Pandas DataFrame
df = pd.read_csv(filename)
```

In [ ]:
```
# Look at the data frame
df.head()
```

Out[ ]:

| | Id | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm | Species |
|---|---|---|---|---|---|---|
| **0** | 1 | 5.100 | 3.500 | 1.400 | 0.200 | Iris-setosa |
| **1** | 2 | 4.900 | 3.000 | 1.400 | 0.200 | Iris-setosa |
| **2** | 3 | 4.700 | 3.200 | 1.300 | 0.200 | Iris-setosa |
| **3** | 4 | 4.600 | 3.100 | 1.500 | 0.200 | Iris-setosa |
| **4** | 5 | 5.000 | 3.600 | 1.400 | 0.200 | Iris-setosa |

# WORKFLOW: Clean and Preprocess the Dataset

# STEP 3: Clean the data

- Find and Mark Missing Values
- If there are no missing data points, then proceed to Step 4.

In [ ]:
```
#Add your codes here
# mark zero values as missing or NaN
df[[ 'SepalLengthCm' , 'SepalWidthCm' , 'PetalLengthCm' ,'PetalWidthCm' ]] \
= df[['SepalLengthCm' , 'SepalWidthCm' ,'PetalLengthCm' , 'PetalWidthCm'
]].replace(0,np.NaN)

# count the number of NaN values in each column
print(df.isnull().sum())
```

```
Id               0
SepalLengthCm    0
SepalWidthCm     0
PetalLengthCm    0
PetalWidthCm     0
Species          0
dtype: int64
```

# STEP 4: Performing the Exploratory Data Analysis (EDA)

- Print a count of the number of rows (observations) and columns (variables)
- Print the data types of all variables
- Print a summary statistics of the data
- Print the number of records in each class

In [ ]:
```
#Add your codes here

# Get the number of records/rows and the number of variables/columns
print("Shape of the dataset(rows, columns):",df.shape)
```

```
Shape of the dataset(rows, columns): (150, 6)
```

In [ ]:
```
#get the data types of all the variables / attributes in the data set
print(df.dtypes)
```

```
Id                int64
SepalLengthCm    float64
SepalWidthCm     float64
PetalLengthCm    float64
PetalWidthCm     float64
Species           object
dtype: object
```

In [ ]:
```
#return the summary statistics of the numeric variables/attributes in the data set
print(df.describe())
```

```
             Id  SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
count 150.000        150.000       150.000        150.000       150.000
mean   75.500          5.843         3.054          3.759         1.199
std    43.445          0.828         0.434          1.764         0.763
min     1.000          4.300         2.000          1.000         0.100
25%    38.250          5.100         2.800          1.600         0.300
50%    75.500          5.800         3.000          4.350         1.300
75%   112.750          6.400         3.300          5.100         1.800
max   150.000          7.900         4.400          6.900         2.500
```

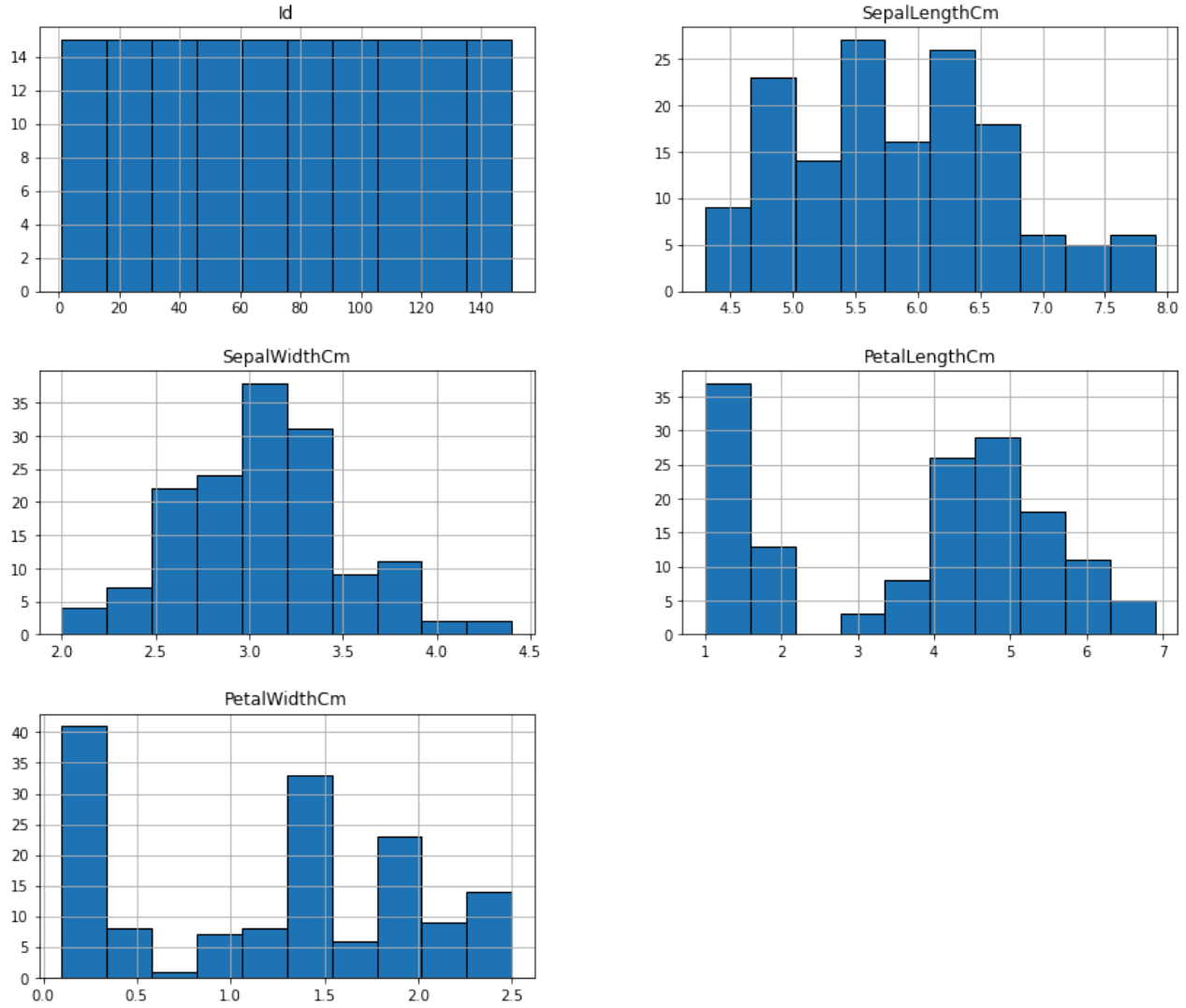
In [ ]:
```
#Add your codes here

#class distribution i.e. how many records are in each class
print(df.groupby('Species').size())
```

```
Species
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
dtype: int64
```

# STEP 4A: Create Histograms

- Create histograms from the dataframe df that is black with a figure size of 14 x 12
- Plot the histograms

In [ ]:
```
#Add your codes here

# Plot histogram for each variable.
df.hist(edgecolor= 'black',figsize=(14,12))
plt.show()
```

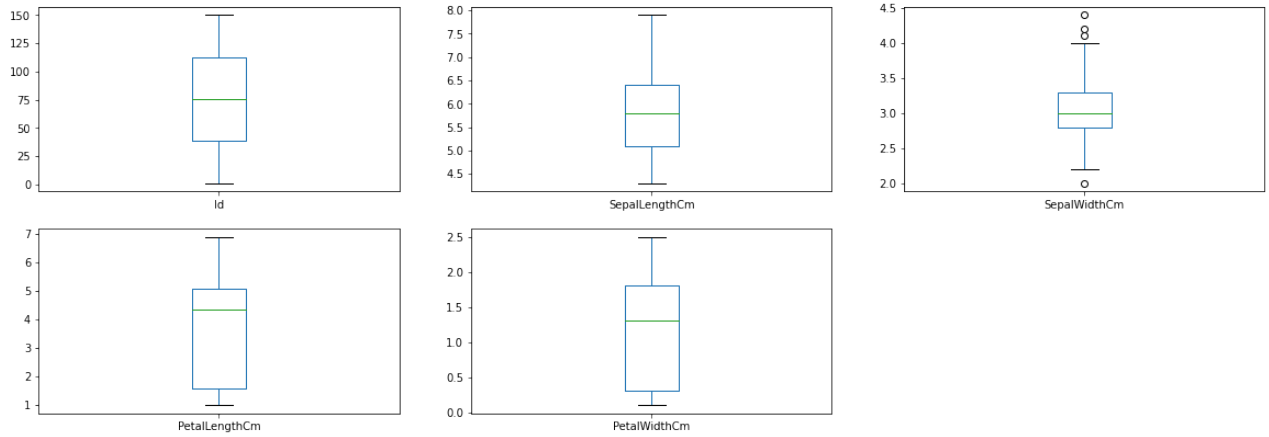



# Step 4B: Creating Boxplots

- Create boxplots from the dataframe df with a layout (5,3) and figure size (20,18). Ensure subplots is True and sharex is False.
- Plot the boxplots

In [ ]:

```
#Add your codes here

# Boxplots
df.plot(kind="box", subplots=True, layout=(5,3), sharex=False, figsize=(20,18))
plt.show()
```




# Step 4C: Create Pair Plots

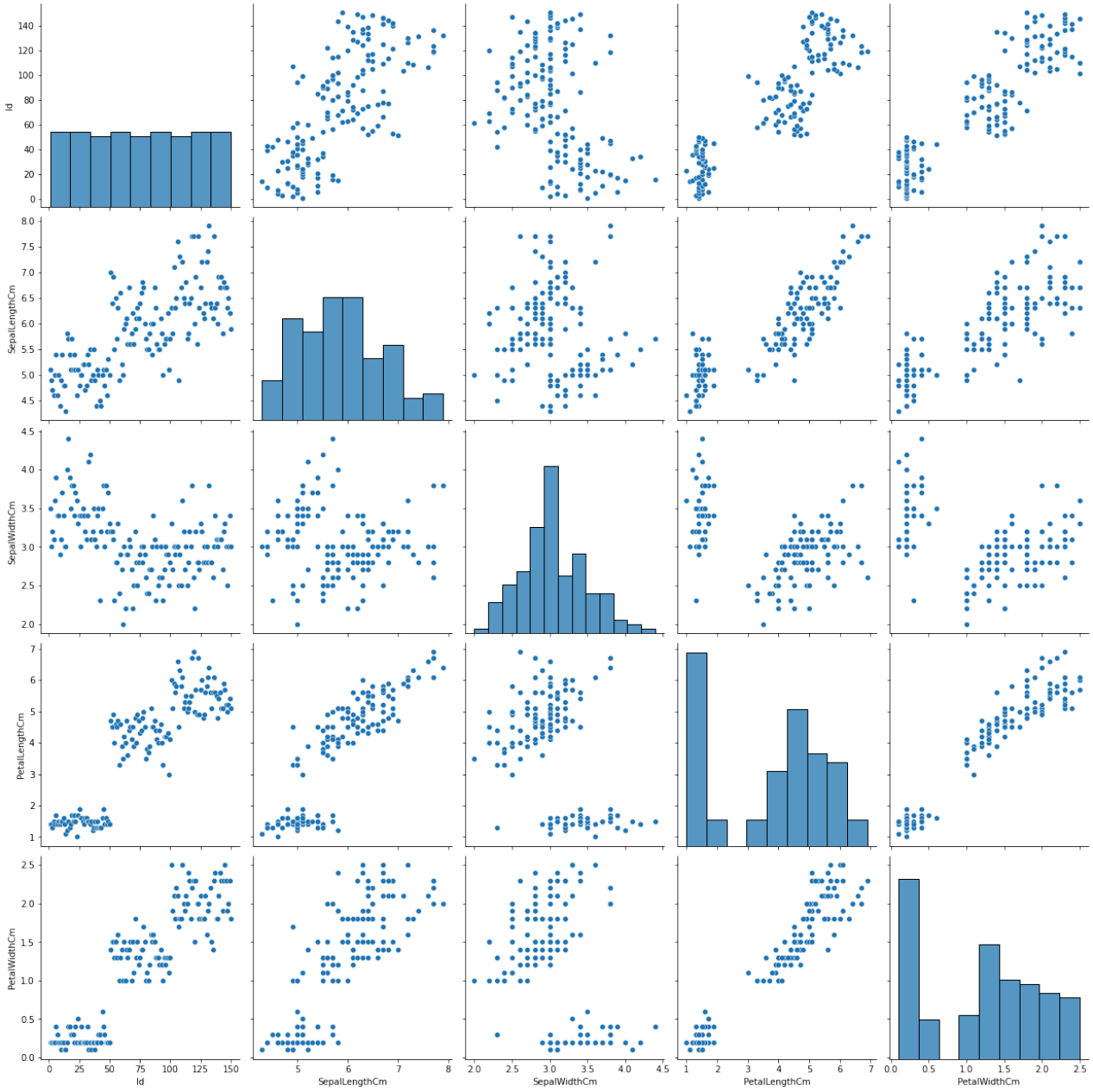
- Create pair plots of the dataframe with a height of 3.5
- Plot the pair plots
- Add color

Note: Please click on the above URL to learn more about Pair Plots

https://seaborn.pydata.org/generated/seaborn.pairplot.html

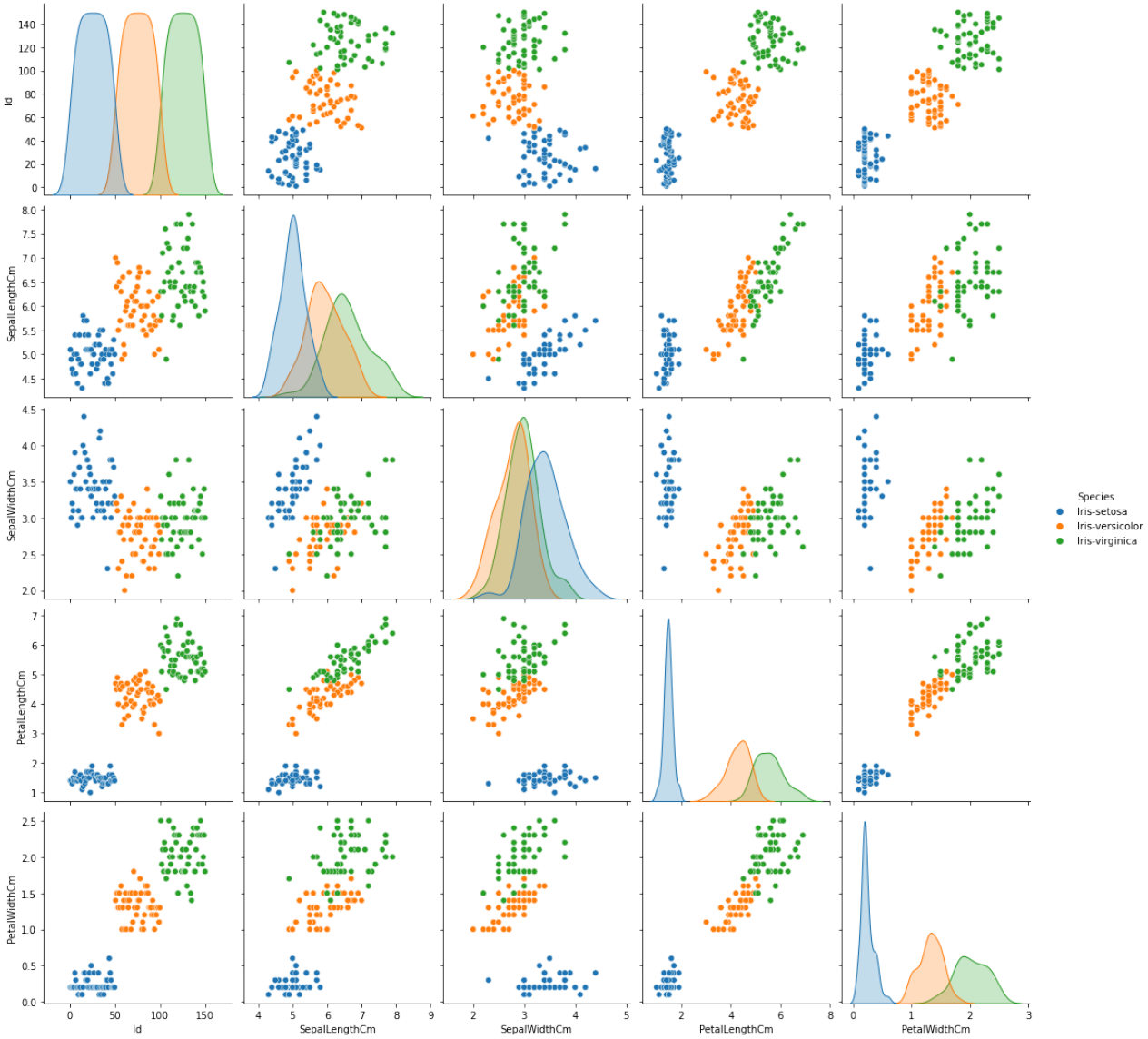
In [ ]:

```
#Add your codes here

# Create pair plots
sns.pairplot(df, height=3.5);
plt.show()
```




In [ ]:

```
#Add your codes here

# Let's try that again using color. Notice: assigning a hue variable adds a semantic mapp

sns.pairplot(df, hue='Species', height=3, aspect= 1);
```




# Step 4D: Creating Violin Plots

- Create violin plots
- Plot the violin plots
- Add color

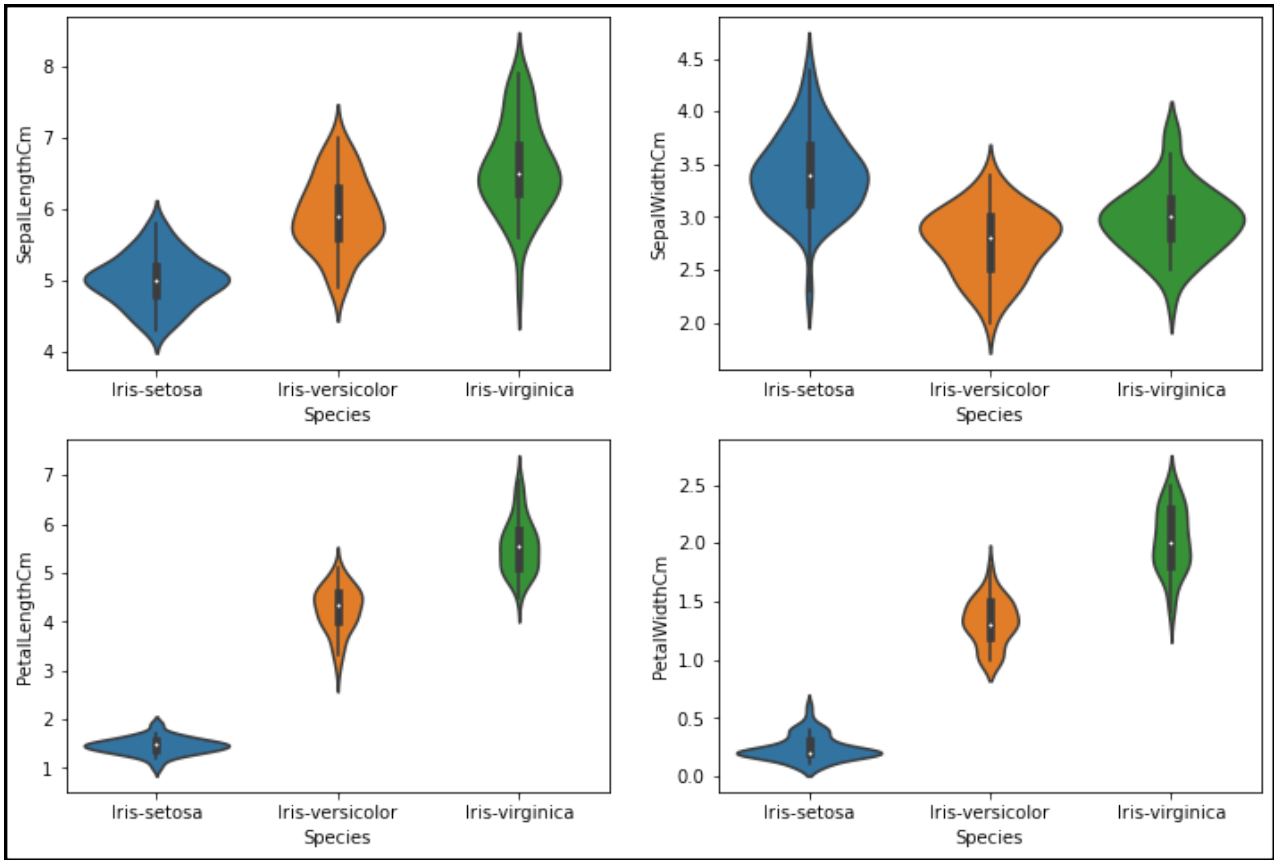
Note: Please click on the above URL to learn more about Violin Plots

https://seaborn.pydata.org/generated/seaborn.violinplot.html

In [ ]:

```
#Add your codes here
# Violin Plot
plt.figure(edgecolor="black", linewidth= 1.2,figsize=(12,8));
plt.subplot(2,2,1)
sns.violinplot(x='Species', y = 'SepalLengthCm', data=df)
plt.subplot(2,2,2)
sns.violinplot(x='Species', y = 'SepalWidthCm', data=df)
plt.subplot(2,2,3)
sns.violinplot(x='Species', y = 'PetalLengthCm', data=df)
```

```
plt.subplot(2,2,4)
sns.violinplot(x='Species', y = 'PetalWidthCm', data=df);
```




# WORKFLOW: DATA SPLIT

## STEP 5: Separate the Dataset into Input & Output NumPy Arrays

- Store the dataframe values into a NumPy array
- Separate the array into input and output components by slicing

In [ ]:

```
# store dataframe values into a numpy array
array = df.values
# separate array into input and output by slicing
# for X(input) [:, 1:5] --> all the rows, columns from 1 - 5
# these are the independent variables or predictors
X = array[:,1:5]
# for Y(input) [:, 5] --> all the rows, column 5
# this is the value we are trying to predict
Y = array[:,5]
```

## STEP 6: Split into Input/Output Array into Training/Testing Datasets

- Split the dataset into training at 67% and test at 33% with the seed = 7

In [ ]:
```
# split the dataset --> training sub-dataset: 67%; test sub-dataset: 33%
test_size = 0.33
#selection of records to include in each data sub-dataset must be done randomly
seed = 7
```

In [ ]:
```
#Add your codes here
#split the dataset (input and output) into training / test datasets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=test_size,random_stat
```

# WORKFLOW: TRAIN MODEL

# STEP 7: Build and Train the Model

- Assign LogisticRegression to the model
- Train the model
- Print the classification report

In [ ]:
```
#Add your codes here
#build the model
model = LogisticRegression(random_state=seed, max_iter=1000)
# train the model using the training sub-dataset
model.fit(X_train, Y_train)
```

Out[ ]:
```
LogisticRegression(max_iter=1000, random_state=7)
```

In [ ]:
```
#Add your codes here
#print the classification report
predicted = model.predict(X_test)

report = classification_report(Y_test, predicted)

print("Classification Report: ", "\n", "\n",report)
```

```
Classification Report:

                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        14
Iris-versicolor       0.89      0.89      0.89        18
 Iris-virginica       0.89      0.89      0.89        18

       accuracy                           0.92        50
      macro avg       0.93      0.93      0.93        50
   weighted avg       0.92      0.92      0.92        50
```

# WORKFLOW: SCORE MODEL 1

# STEP 8: Score the Accuracy of the Model

- Calculate accuracy score
- Print the score

In [ ]:
```
#Add your codes here
#score the accuracy level
result = model.score(X_test, Y_test)
#print out the results
print(("Accuracy: %.3f%%") % (result*100.0))
```

```
Accuracy: 92.000%
```

# Step 9: Prediction

- Execute model prediction

Note: We have now trained the model and using that trained model to predict the type of flower we have with the listed values for each variable.

In [ ]:
```
#Add your codes here

#predict with[5.3, 3.0, 4.5, 1.5]
model.predict([[5.3, 3.0, 4.5, 1.5]])
```

Out[ ]:
```
array(['Iris-versicolor'], dtype=object)
```

In [ ]:
```
#Add your codes here
#predict with [5, 3.6, 1.4, 1.5]
model.predict([[5, 3.6, 1.4, 1.5]])
```

Out[ ]:
```
array(['Iris-setosa'], dtype=object)
```

# WORKFLOW: EVALUATE MODELS

# Step 10: Train & Score Model 2 Using K-Fold Cross Validation Data Split

- Specify the k-size to 10
- Fix the random seed to 7
- Split the entire data set
- Obtain the accuracy level
- Train the model and run K-fold cross-validation
- Print results

In [ ]:
```
#Add your codes here

# Evaluate the algorithm and specify the number of times of repeated splitting, in this c
n_splits=10
#Fix the random seed. You must use the same seed value so that the same subsets can be ob
```

```
seed=7

# Split the whole data set into folds
kfold=KFold(n_splits, random_state=seed, shuffle=True)

# for logistic regression, we can use the accuracy level to evaluate the model
scoring="accuracy"
```

In [ ]:

```
#Add your codes here

#train the model and run K-fold cross-validation to validate / evaluate the model
results=cross_val_score (model, X, Y, cv=kfold, scoring=scoring)

# print the evaluation results. The result is the average of all the results obtained fro
print("Accuracy: %.3f (%.3f)"% (results.mean(), results.std()))
```

```
Accuracy: 0.967 (0.054)
```

# GREAT JOB! YOU ARE DONE.