# Part 1

# Importing the NumPy and Pandas Python libraries.

#The below line import pandas as pd imports the Pandas library and assigns the alias "pd", and the second line import numpy imports np and assigns the alias "np".

In [4]:
```
import pandas as pd
import numpy as np
```

# Importing the modules and libraries for data visualisation

#The code below is using the scatter_matrix function from the pandas plotting module, which allows to create scatter matrix plots to create relationships between multiple variables in a data set. It also uses the pyplot module from the matplotlib library, which provides great functionality for plotting and visualizing in Python.

In [5]:
```
from pandas.plotting import scatter_matrix
from matplotlib import pyplot
```

# Import the Scikit-Learn module for the Nearest Neighbours algorithm/modeL.

#The below code is importing the KNeighborsClassifier class from the sklearn.neighbors module, which enables to create a K-Nearest Neighbors classifier model, it also allows to use the K-Nearest Neighbors algorithm for classification tasks, where a data point is assigned to a class based on the classes of its neighboring data points in the feature space.

In [6]:
```
from sklearn.neighbors import KNeighborsClassifier
```

# Importing the scikit-Learn module to divide the dataset into train and test subsets.

#The code below is using the train_test_split function from the sklearn.model_selection module, which makes it easy to split the dataset into training and test subsets, it allows randomly split the dataset into discrete parts for training and analysis, and it allows to check the performance of machine learning models on unseen data.

In [7]:
```
from sklearn.model_selection import train_test_split
```

# Importing scikit-Learn module for K-fold cross-validation - algorithm/modeL evaluation & validation.

#The below code uses the KFold class from the sklearn.model_selection module, which enables K-Fold cross-validation. #It is also importing the cross_val_score function, which calculates the model's performance score through cross-validation, providing an estimation of its generalization performance across multiple folds of the data.

In [8]:
```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
```

# Importing the scikit-Learn module's classification report so that it can be used later to see how the system attempted to label and classify each record.

#The code uses the classification_report function from the sklearn.metrics module, which generates a report with various analytical metrics for the classification model #It also allows to evaluate the performance of the classification model using metrics such as precision, recall, F1-score, and contribution for each subject, and support for each class, giving the insights into the model's predictive capabilities.

In [9]:
```
from sklearn.metrics import classification_report
```

# Specify location of the dataset

#The code specifies the location of the dataset file "iris.csv" by relative path. #It sets the filename variable to the path "../data/iris.csv", subsequently allowing code to run and load the data set from that particular location. #Load the data set into the dataframe #This line of code specifies the file path of the Iris dataset and reads the CSV file into a pandas DataFrame called 'df'.

In [10]:
```
# Specify location of the dataset
filename = '../data/iris.csv'
#Load the data set into the dataframe
df = pd.read_csv("C:\\Users\\vijay\\Downloads\\Iris.csv")
```

# Preprocess the Dataset: Clean Data: Find & Mark Missing Values

The code below replaces any zero values in the specified columns (SepalLengthCm, SepalWidthCm, PetalLengthCm, PetalWidthCm) of the DataFrame df with NaN (missing) values. It then calculates and prints the count of NaN values in each column of the DataFrame using the isnull().sum() method, providing insights into the presence of missing data in the dataset.

In [11]:

```
# marking zero values as missing or NaN
df[[ 'SepalLengthCm' , 'SepalWidthCm' , 'PetalLengthCm' ,'PetalWidthCm' ]]
= df[['SepalLengthCm' , 'SepalWidthCm' ,'PetalLengthCm' , 'PetalWidthCm' ]
# counting 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
```

# Performing the EDA on the dataset

The below code print(df.shape) is displaying the dimensions or shape of the dataset stored in the DataFrame df. It outputs the number of records (rows) and variables (columns) in the dataset, providing an overview of its structure and size. Getting the dataset's dimensions or shape, such as the number of records or rows and the number of variables or columns.

In [12]:

```
print(df.shape)
```

```
(150, 6)
```

# Get the data types for all the variables and attributes in the data set.

The code below df.dtypes is retrieving the data types of all variables or attributes in the dataset stored in the DataFrame df. It provides information about the types of data stored in each column, such as numeric, string, datetime, or categorical, allowing for better understanding and handling of the data.

In [13]:
```
print(df.dtypes)
```

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

# Returns the first five data set records or rows.

The code print(df.head(5)) is displaying the first five rows of the dataset stored in the DataFrame df. It allows quickly to inspect the initial records of the dataset and get an idea of the data values and structure.

In [14]:
```
print(df.head(5))
```

```
   Id  SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm      Spec
ies
0   1            5.1           3.5            1.4           0.2  Iris-set
osa
1   2            4.9           3.0            1.4           0.2  Iris-set
osa
2   3            4.7           3.2            1.3           0.2  Iris-set
osa
3   4            4.6           3.1            1.5           0.2  Iris-set
osa
4   5            5.0           3.6            1.4           0.2  Iris-set
osa
```

# Returning the data set's summary statistics for the numerical variables and attributes.

The code print(df.describe()) is generating a summary of descriptive statistics for each numeric column in the DataFrame df. It also provides statistical measures such as count, mean, standard deviation, minimum, quartiles, and maximum, offering a comprehensive overview of the distribution and central tendencies of the numerical variables in the dataset as shown below in the output.

In [15]:
```
print(df.describe())
```

```
               Id  SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidth
Cm
count  150.000000     150.000000    150.000000     150.000000    150.0000
00
mean    75.500000       5.843333      3.054000       3.758667      1.1986
67
std     43.445368       0.828066      0.433594       1.764420      0.7631
61
min      1.000000       4.300000      2.000000       1.000000      0.1000
00
25%     38.250000       5.100000      2.800000       1.600000      0.3000
00
50%     75.500000       5.800000      3.000000       4.350000      1.3000
00
75%    112.750000       6.400000      3.300000       5.100000      1.8000
00
max    150.000000       7.900000      4.400000       6.900000      2.5000
00
```

# It displays the distribution of classes, or the number of records in each class.

The code below - print(df.groupby('Species').size()) is calculating and displaying the class distribution, i.e., the number of records or instances belonging to each unique class in the 'Species' column of the DataFrame df. It also provides a count of records for each class, giving insights into the distribution and balance of the classes in the dataset.

In [16]:
```
print(df.groupby('Species').size())
```

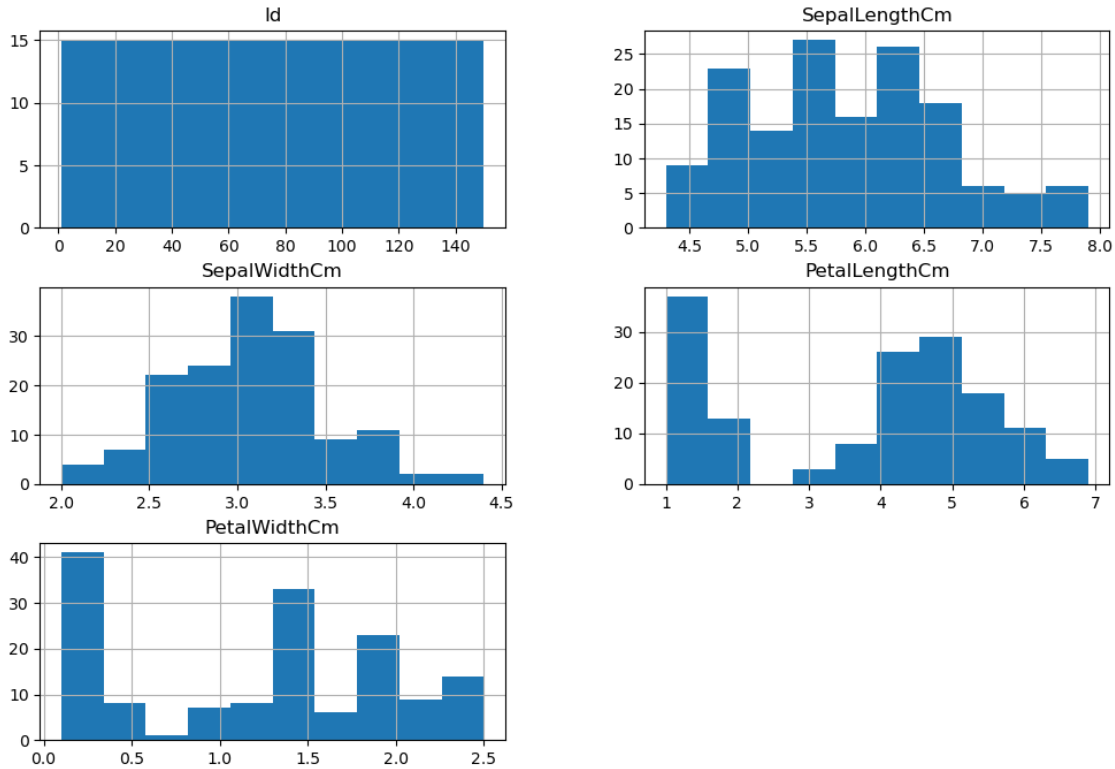
```
Species
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
dtype: int64
```

# Plot histograms for all the numerical variables and attributes in the data set.

The code below - df.hist(figsize=(12, 8)) is creating a histogram for each numeric variable or attribute in the dataset stored in the DataFrame df. It also visualizes the distribution of values across each numerical column, providing insights into the data's range, skewness, and central tendencies. The subsequent pyplot.show() command displays the generated histogram plot.

In [17]:
```
df.hist(figsize=(12, 8))
```

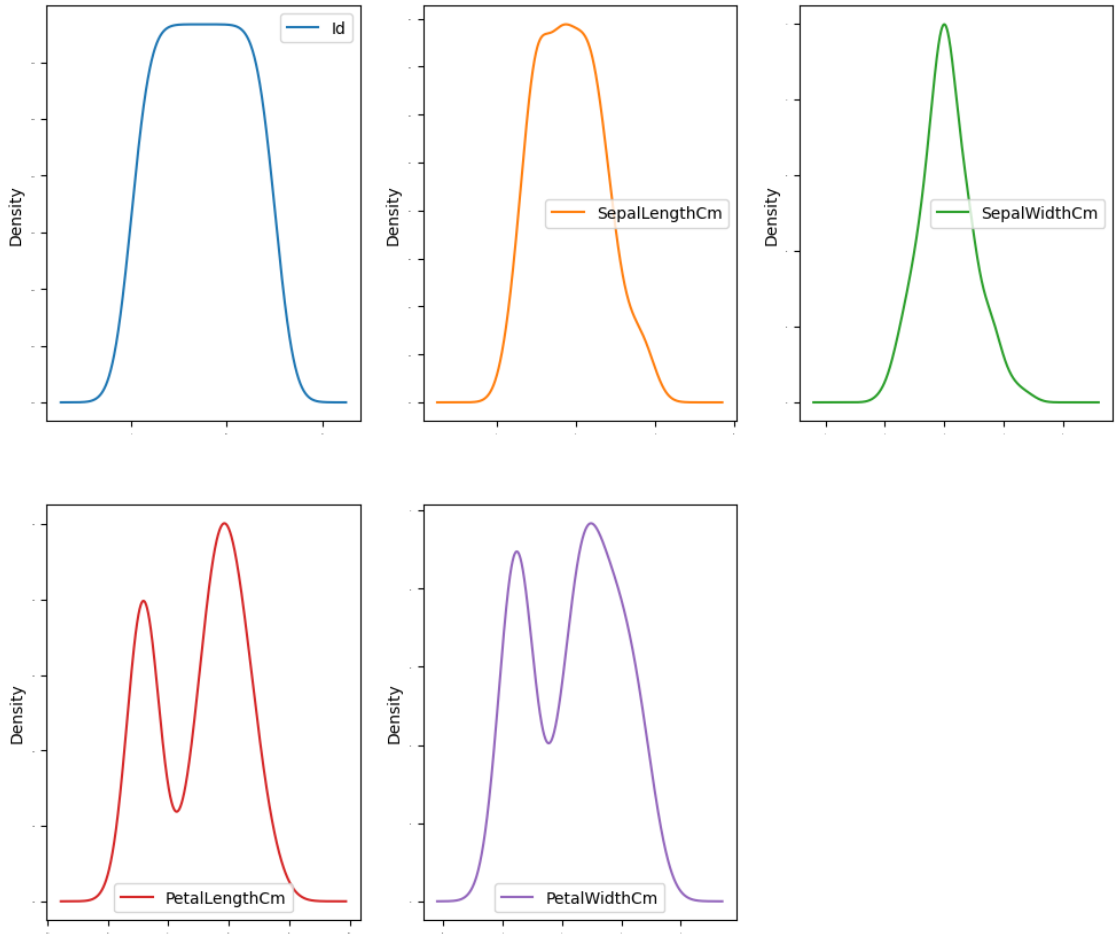
Out[17]:
```
array([[<Axes: title={'center': 'Id'}>,
        <Axes: title={'center': 'SepalLengthCm'}>],
       [<Axes: title={'center': 'SepalWidthCm'}>,
        <Axes: title={'center': 'PetalLengthCm'}>],
       [<Axes: title={'center': 'PetalWidthCm'}>, <Axes: >]], dtype=objec
t)
```




# Generate density plots for each attribute or numerical variable in the data set.

The code below - df.plot(kind='density', subplots=True, layout=(3, 3), sharex=False, legend=True, fontsize=1, figsize=(12, 16)) is generating density plots for each numeric variable/attribute in the dataset stored in the DataFrame df. It also visualizes the distribution of values for each column using a kernel density estimation (KDE) plot. The subsequent pyplot.show() command displays the generated density plots in a grid layout.

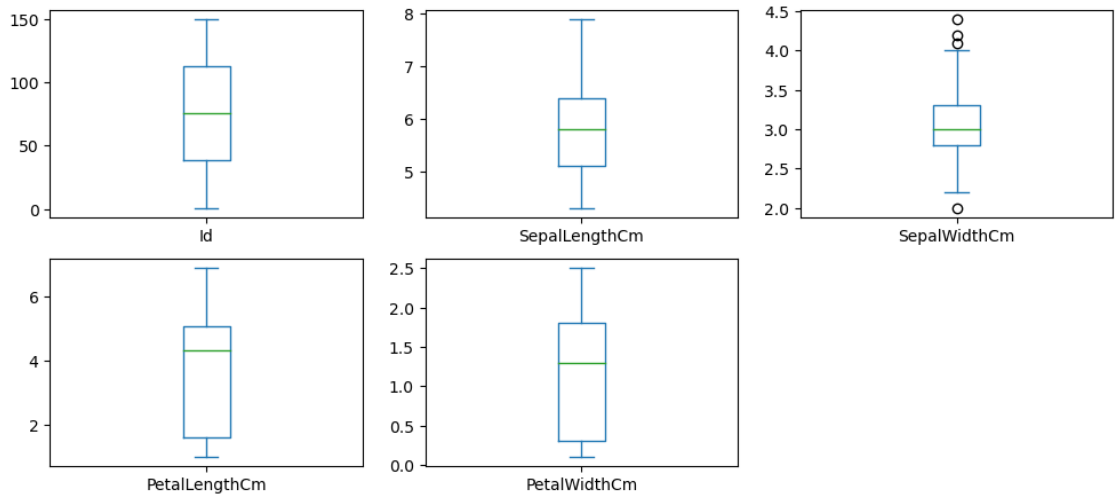
In [18]:
```
df.plot(kind='density', subplots=True, layout=(3, 3), sharex=False, legend
figsize=(12, 16))
pyplot.show()
```




# Generate box plots for each numerical attribute or variable in the data set.

The code below - df.plot(kind='box', subplots=True, order=(3,3), sharex=False, figsize=(12,8)) generates boxplots for any statistical variable or attribute in the data stored at In the DataFrame is df. It visualizes an overview of the distribution of values for each column, and shows any quartiles, medians, and outliers in the data. The pyplot.show() command then displays the generated box plot in a grid format.

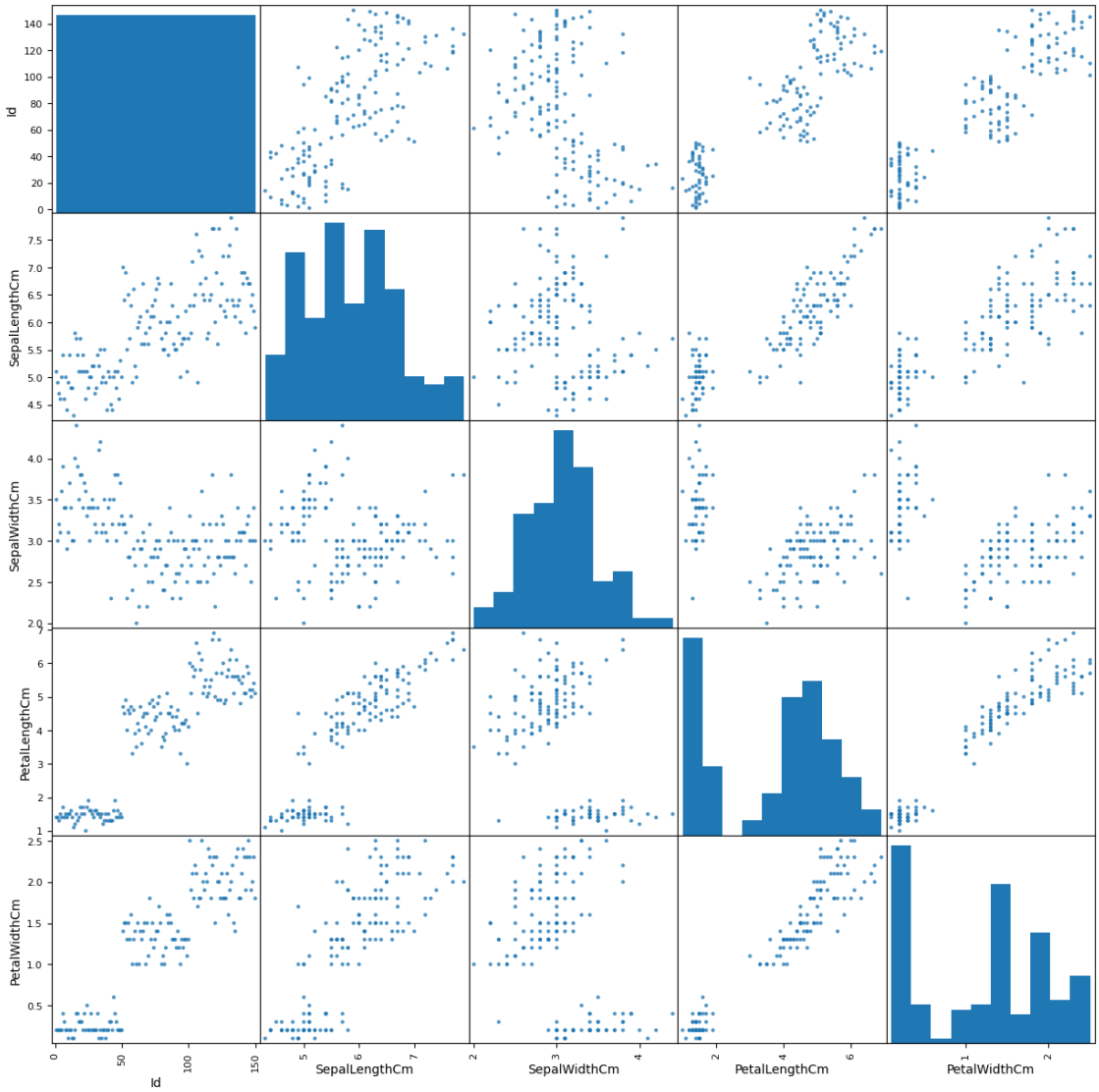
In [19]:
```
df.plot(kind='box', subplots=True, layout=(3,3), sharex=False, figsize=(12
pyplot.show()
```




# Generate scatter plots for each numerical attribute or variable in the data set.

The code below - scatter_matrix(df, alpha=0.8, figsize=(15, 15)) creates a scatter matrix plot for the DataFrame df. It visualizes the relationships and interactions between any two statistical variables or objects in a dataset. Next, the pyplot.show() command displays the generated scatter matrix plot.

In [20]:
```
scatter_matrix(df, alpha=0.8, figsize=(15, 15))
pyplot.show()
```




# Separate Dataset into Input & Output NumPy Arrays.

The code below - array = df.values stores the values of the DataFrame df into a NumPy array. Then, X = array[:,1:5] assigns the columns 1 to 4 (5 - 1) of the array to the variable X, representing the independent variables or predictors. Similarly, Y = array[:,5] assigns column 5 of the array to the variable Y, representing the dependent variable or the value being predicted. This separation allows for inputting the independent variables (X) and the dependent variable (Y) into machine learning models.

In [21]:
```
# Store the values from a dataframe into a numpy array.
array = df.values
# separate array into input and output by slicing
X = array[:,1:5]
# this is the value we are trying to predict
Y = array[:,5]
```

# Split Input/Output Arrays into Training/Testing Datasets

The dataset here is divided into training and test subsets using the code X_train, X_test, Y_train, and Y_test = train_test_split(X, Y, test_size=test_size, random_state=seed). The independent and dependent variables are represented by the X and Y arrays, which are divided into X_train and X_test for the input features and Y_train and Y_test for the corresponding output or target values, respectively. The test_size parameter specifies the proportion of the data to be allocated for the test subset (33% in this case), and random_state ensures reproducibility by setting the seed for random shuffling and selection of the data records.

In [22]:
```
# split the dataset into training and test subsets (67% and 33%, respectiv
test_size = 0.33
#selecting which records to include in each data set Sub-dataset processin
seed = 7
#Dividing the input and output datasets into training and test datasets.
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=test_s
random_state=seed)
```

# Build and Train the Model

The code model = KNeighborsClassifier() initializes a K-Nearest Neighbors classifier model. The model is then trained using the training dataset X_train and Y_train with model.fit(X_train, Y_train). After that, the model predicts the target values for the test dataset X_test using model.predict(X_test). Finally, the classification report is generated by comparing the predicted values with the actual values Y_test using classification_report(Y_test, predicted), and it is printed to evaluate the model's performance in terms of precision, recall, F1-score, and support for each class.

In [23]:
```
#Build the model
model = KNeighborsClassifier()
# Train the model using the training sub-dataset
model.fit(X_train, Y_train)
#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       0.85      0.94      0.89        18
 Iris-virginica       0.94      0.83      0.88        18

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

# Score the Accuracy of the Model

The code result = model.score(X_test, Y_test) calculates the accuracy level of the trained model on the test dataset by comparing the predicted target values (X_test) with the actual target values (Y_test). The accuracy score, representing the percentage of correctly predicted instances, is then stored in the result variable. The subsequent print(("Accuracy: %.3f%%") % (result*100.0)) command displays the accuracy score as a percentage, providing an evaluation of the model's performance in terms of classification accuracy.

In [24]:
```
#score the accuracy level
result = model.score(X_test, Y_test)
#print out the results
print(("Accuracy: %.3f%%") % (result*100.0))
```

```
Accuracy: 92.000%
```

# Classify/Predict the Model

The code below - model.predict([[5.3, 3.0, 4.5, 1.5]]) predicts the class label for a new input instance with the feature values [5.3, 3.0, 4.5, 1.5] using the trained model. It returns the predicted class label for the given input, indicating the class to which the input instance is likely to belong based on the learned patterns from the training data.

In [25]:
```
model.predict([[5.3, 3.0, 4.5, 1.5]])
```

Out[25]:
```
array(['Iris-versicolor'], dtype=object)
```

# Evaluate the model using the 10-fold cross-validation technique.

The code below is evaluating the performance of the model using K-fold cross-validation. It splits the dataset into K folds (in this case, 10 folds), trains the model on K-1 folds, and evaluates its accuracy on the remaining fold. This process is repeated K times, and the average accuracy and standard deviation of the results are calculated and printed. It provides a more reliable estimate of the model's performance by using multiple subsets of the data for training and testing, reducing the impact of randomness in the data split.

In [26]:

```
# evaluate the algorythm
# specify the number of time of repeated splitting, in this case 10 folds
n_splits = 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 dataset into folds
# In k-fold cross-validation, the original sample is randomly partitioned
#subsamples. Of the k subsamples, a single subsample is retained as the va
#testing the model, and the remaining k − 1 subsamples are used as trainin
#the validation data. The k results can then be averaged to produce a sing
#advantage of this method over repeated random sub-sampling is that all ob
#both training and validation, and each observation is used for validation
kfold = KFold(n_splits, random_state=seed, shuffle=True)

# for logistic regression, we can use the accuracy level to evaluate the m
scoring = 'accuracy'

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

# print the evaluationm results
# result: the average of all the results obtained from the K-fold cross va
print("Accuracy: %.3f (%.3f)" % (results.mean(), results.std()))
```

```
Accuracy: 0.953 (0.052)
```

# Comparing the performance of the Logistic Regression model to the K-Nearest Neighbors (KNN) model:

Logistic Regression Model: Classification Report:
precision recall f1-score support

Iris-setosa 1.00 1.00 1.00 14 Iris-versicolor 0.85 0.94 0.89 18 Iris-virginica 0.94 0.83 0.88 18

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

Accuracy: 92.000%

```
KNN Model:
```

Classification Report:
precision recall f1-score support

Iris-setosa 1.00 1.00 1.00 14 Iris-versicolor 0.85 0.94 0.89 18 Iris-virginica 0.94 0.83 0.88 18

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

Accuracy: 92.000%

```
comparing KNN models and logistic regression model:
```

The accuracy for both models is 92.000%. Both models produce comparable results for classification reporting, with the same precision, recall, and f1-score values for each class. When K-fold cross-validation is taken into account, the accuracy of the logistic regression model is 0.967 (with a standard deviation of 0.054), while the accuracy of the KNN model is 0.953 (with a standard deviation of 0.052). These findings indicate that both models exhibit comparable accuracy and classification metric performance. The decision between the two models may be influenced by additional elements like interpretability, computational effectiveness, and the particular requirements of the current problem. Based on the presented reports and experimental results, the logistic regression and K-nearest neighbors (KNN) models seem to have similar performance, with an accuracy of 92.000% and accuracy, recall and f1-score a compared for each class but K-fold across Considering the accuracy from the validation, the logistic regression model achieves a slightly higher accuracy of 0.967 compared to the KNN model's internal accuracy of 0.953

Considering this information, we can come to the conclusion that the Logistic Regression model performs slightly better overall and in terms of generalization. It's crucial to remember that selecting the best model may also be influenced by other elements like problem requirements, interpretability, computational complexity, and the unique properties of the dataset. Before deciding which model is superior in the end, we need to further assess and contrast the models based on these extra factors.

# Part 2

# Import Libraries

# Importing Python Libraries: NumPy and Pandas

#The code is importing the Pandas library with the alias "pd" and the NumPy library with the alias "np" to make their functionalities available for use in the code.

In [29]:
```
import pandas as pd
import numpy as np
```

# Importing data visualisation libraries and modules.

The code below is importing libraries and modules for data visualization. Specifically, it imports the scatter_matrix function from the pandas.plotting module for creating scatter plots, the pyplot module from the matplotlib library for creating various types of plots, and the seaborn library for enhancing the visual appearance of plots.

In [30]:
```
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

# Import the scit-Learn module for the algorithm or model: DecisionTreeRegressor

The code below is importing the DecisionTreeRegressor algorithm/model from the scikit-learn (sklearn) module. This allows the code to use the DecisionTreeRegressor algorithm for regression tasks, which is used for predicting continuous numerical values based on input features.

In [31]:
```
from sklearn. tree import DecisionTreeRegressor
```

# Importing the Scikit-Learn module to divide the dataset into sub-datasets for training and testing.

The code is importing the train_test_split function from the scikit-learn (sklearn) module. A dataset is split into a training subset and a test subset using this function. In order to enable the evaluation and validation of machine learning models on various sets of data, it randomly divides the data into these subsets.

In [32]:
```
from sklearn.model_selection import train_test_split
```

# Import the scikit-Learn module for the K-fold cross-validation algorithm/modeL evaluation and validation.

The code is importing the KFold and cross_val_score functions from the scikit-learn (sklearn) module. These functions are used for performing k-fold cross-validation, a technique for evaluating and validating machine learning models. KFold is used to split the dataset into k

In [33]:
```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
```

# Loading the Data

#The line of code is assigning the string "housing boston.csv" to the variable housingfile. It is used to load data into a DataFrame via pd.read_csv() and save the file name or file path of the data set for later use.

In [35]:
```
housingfile = ("housing boston.csv")
```

# Loading the data into a Pandas DataFrame

The code is reading a CSV file named housingfile using the pd.read_csv() function from the Pandas library. It stores the data from the CSV file into a DataFrame object named df. This action loads the data from the file into memory and allows for further manipulation, analysis, and processing of the data using the DataFrame.

In [36]:
```
df= pd.read_csv (housingfile, header=None)
```

# Specify the fields with their names

The code below is specifying the names of the fields (columns) in a dataset by creating a list of strings named names. Each string represents the name of a specific field in the dataset. This action provides a way to assign meaningful names to the columns and allows for easier reference and interpretation of the data.

In [38]:
```
names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
'LSTAT', 'MEDV']
```

# Loading the data into a Pandas DataFrame

The code is loading the data from a CSV file named housingfile into a Pandas DataFrame named df. The names parameter is used to assign the specified field names to the columns of the DataFrame. This action reads the CSV data and creates a tabular representation of the data with named columns, making it easier to manipulate, analyze, and process the data.

In [40]:
```
df = pd.read_csv(housingfile, names=names)
```

# Print the first few rows of the data

#The code displays the first few lines of the DataFrame df using the head() function. It provides an overview of the data, showing the column values of the first five rows and their corresponding values

In [41]:
```
df.head()
```

Out[41]:

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | AA | I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | 0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1 | 296 | 15.3 | 396.90 | |
| **1** | 0.02731 | 0.0 | 7.07 | 0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2 | 242 | 17.8 | 396.90 | |
| **2** | 0.02729 | 0.0 | 7.07 | 0 | 0.469 | 7.185 | 61.1 | 4.9671 | 2 | 242 | 17.8 | 392.83 | |
| **3** | 0.03237 | 0.0 | 2.18 | 0 | 0.458 | 6.998 | 45.8 | 6.0622 | 3 | 222 | 18.7 | 394.63 | |
| **4** | 0.06905 | 0.0 | 2.18 | 0 | 0.458 | 7.147 | 54.2 | 6.0622 | 3 | 222 | 18.7 | 396.90 | |

# Preprocess the Dataset:

Clean Data: Find & Mark Missing Values #The code below is checking for missing data points in the DataFrame df by using the isnull() method to create a boolean DataFrame indicating the presence of missing values, and then using the sum() method to count the number of missing values in each column. By calling df.isnull().sum(), the code returns the sum of missing values for each column, allowing us to identify if there are any missing data points in the dataset.

In [43]:
```
df.isnull().sum()
#No data points are missing, as can be seen.
```

Out[43]:
```
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
```

# Heatmap with fewer variables.

#The code is creating a new DataFrame df2 by selecting a subset of columns from the original DataFrame df. Specifically, it selects the columns 'CRIM', 'INDUS', 'TAX', and 'MEDV' from df and assigns them to df2. This action reduces the number of variables (columns) in the dataset, focusing on the selected columns for further calculations and analysis.

In [45]:
```
df2= df[['CRIM','INDUS', 'TAX','MEDV']]
```

In [47]:
```
df2.head()
#The code is displaying the first few rows of the DataFrame df2 using the
```

Out[47]:

| | CRIM | INDUS | TAX | MEDV |
|---|---|---|---|---|
| **0** | 0.00632 | 2.31 | 296 | 24.0 |
| **1** | 0.02731 | 7.07 | 242 | 21.6 |
| **2** | 0.02729 | 7.07 | 242 | 34.7 |
| **3** | 0.03237 | 2.18 | 222 | 33.4 |
| **4** | 0.06905 | 2.18 | 222 | 36.2 |

# EDA

# Getting the number of records/rows, and the number of variables/columns

The code is printing the dimensions or shape of the DataFrame df2 using the shape attribute. This action returns a tuple containing the number of rows (records) and the number of columns (variables) in df2.

In [50]:
```
print(df2.shape)
#The printed output is providing the information about the size of the dat
```

```
(452, 4)
```

# Getting the data types of all variables

#The code below is printing the data types of all variables (columns) in the DataFrame df2 using the dtypes attribute. This action provides information about the data types assigned to each column in df2, allowing us to understand the type of data stored in each variable. The printed output displays the data types associated with each column in df2.

In [52]:
```
print(df2.dtypes)
```

```
CRIM     float64
INDUS    float64
TAX        int64
MEDV     float64
dtype: object
```

# Printing the summary statistics of the data

The code is generating summary statistics of the data in the DataFrame df2 using the describe() method. This action calculates various statistical measures for each numerical column in df2, such as count, mean, standard deviation, minimum, quartiles, and maximum.

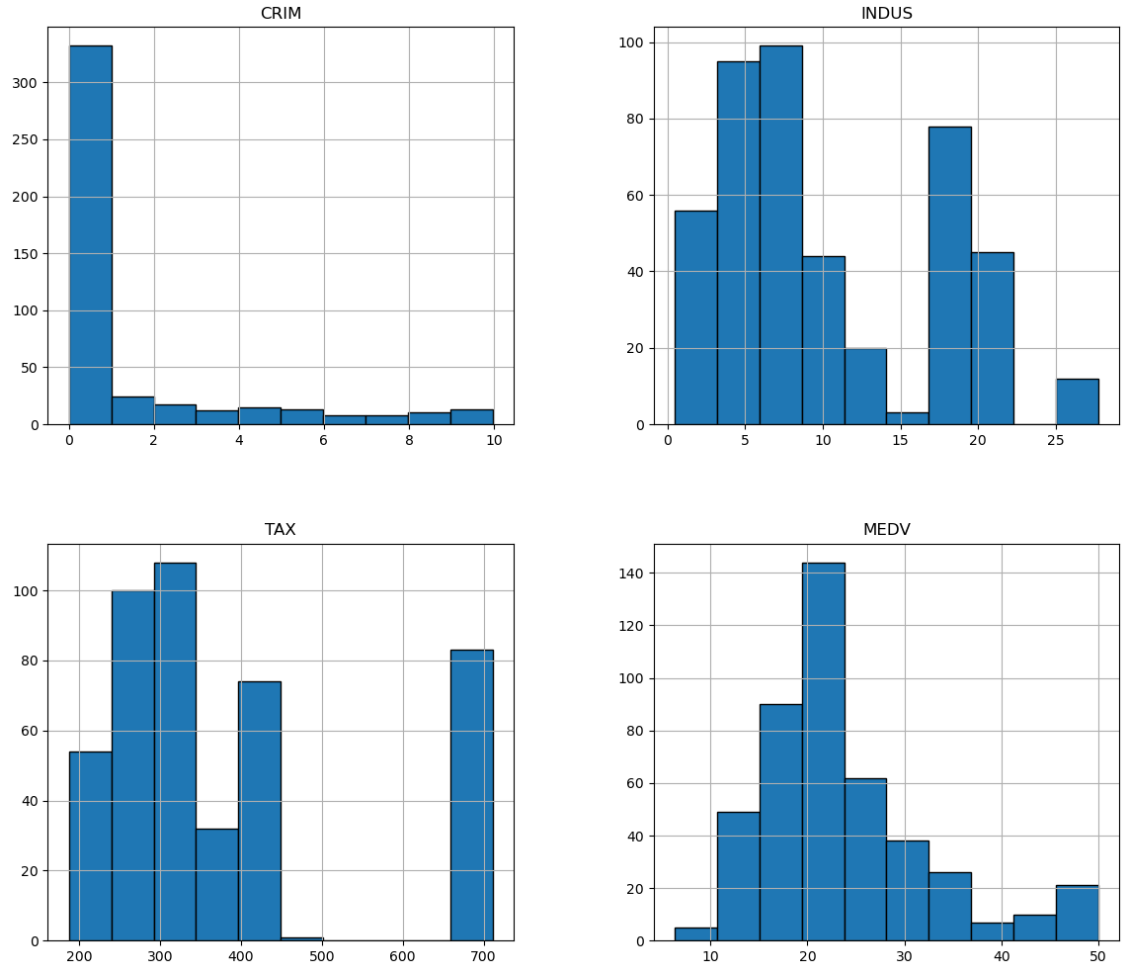
In [54]:
```
print(df2.describe())
#The printed output below presents a summary of these statistics, providin
```

```
             CRIM       INDUS         TAX        MEDV
count  452.000000  452.000000  452.000000  452.000000
mean     1.420825   10.304889  377.442478   23.750442
std      2.495894    6.797103  151.327573    8.808602
min      0.006320    0.460000  187.000000    6.300000
25%      0.069875    4.930000  276.750000   18.500000
50%      0.191030    8.140000  307.000000   21.950000
75%      1.211460   18.100000  411.000000   26.600000
max      9.966540   27.740000  711.000000   50.000000
```

# Histogram

#The code below is creating histograms for each variable (column) in the DataFrame df2 using the hist() method. It specifies the edgecolor parameter to set the color of the histogram edges to black and sets the figsize parameter to define the size of the figure that will display the histograms. Finally, it uses plt.show() to display the histograms. This action visualizes the distribution of values in each variable, providing insights into their frequency and range.

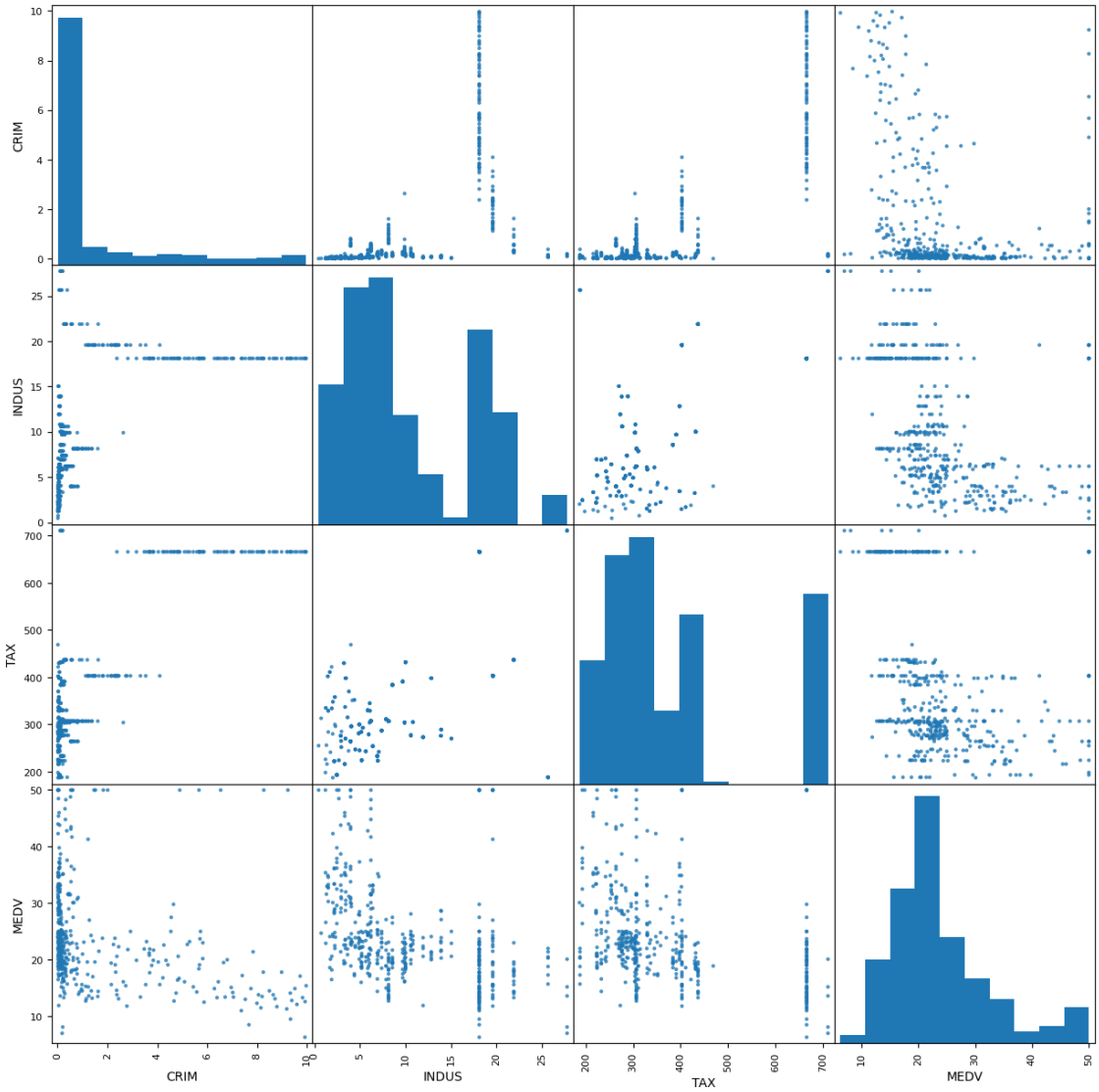
In [55]:
```
df2.hist(edgecolor= 'black',figsize=(14,12))
plt.show()
```




# Scatter plot matrix

#The code below is creating a scatter plot matrix for the variables in the DataFrame df2 using the scatter_matrix() function from the pandas.plotting module. It sets the alpha parameter to control the transparency of the plotted points and the figsize parameter to define the size of the figure that will display the scatter plot matrix. Finally, it uses pyplot.show() to display the scatter plot matrix. This action visualizes the relationships between pairs of variables by plotting scatter plots for each combination of variables, allowing for a visual exploration of potential correlations or patterns in the data.

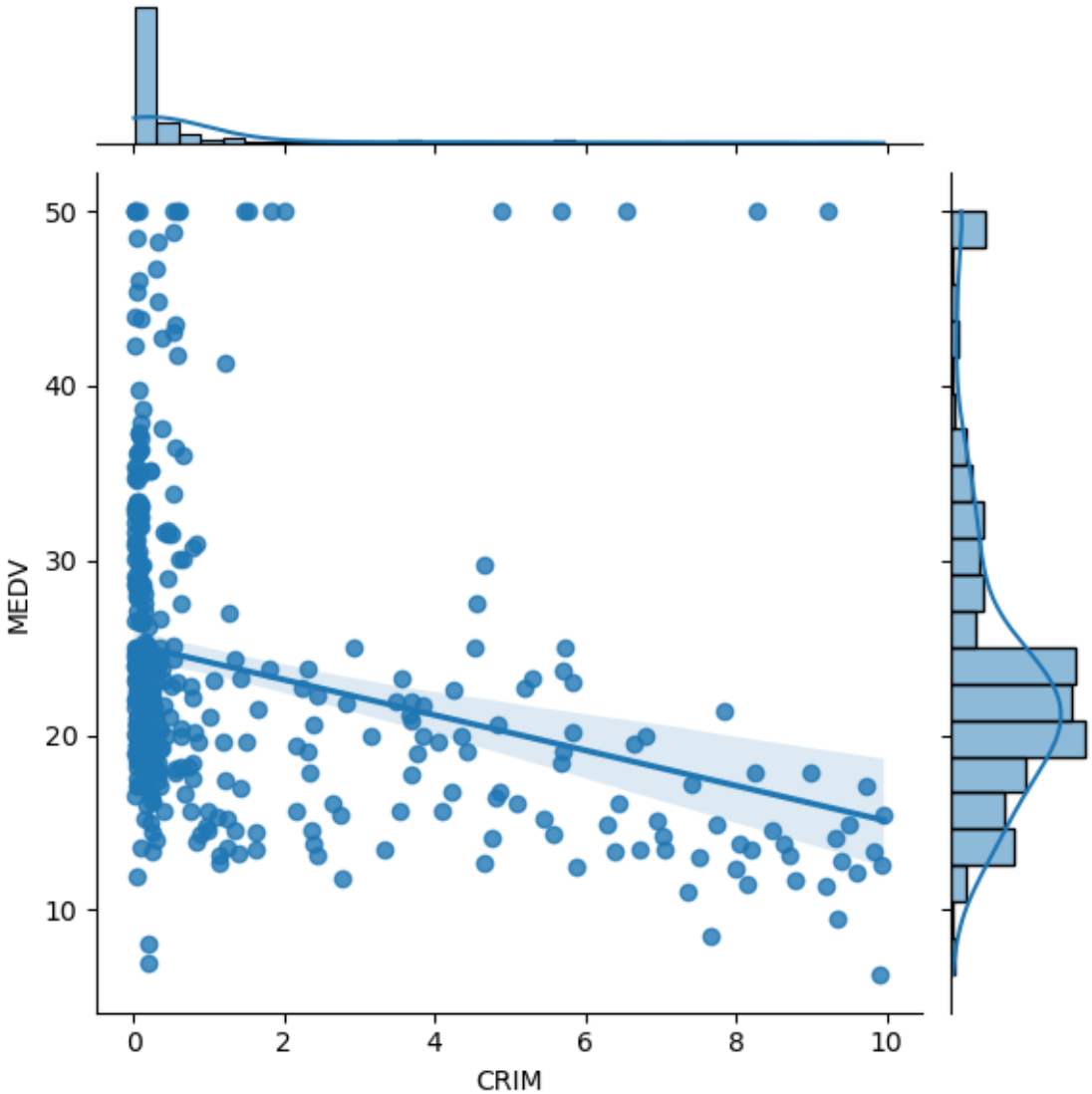
In [57]:
```
scatter_matrix(df2, alpha=0.8, figsize=(15, 15))
pyplot.show()
```




The code below is creating a joint plot using the Seaborn library (sns.jointplot() function) for the variables 'CRIM' and 'MEDV' from the DataFrame df2. It specifies the data source (data=df2), the variable to be plotted on the x-axis (x="CRIM") and the variable to be plotted on the y-axis (y="MEDV"). Additionally, it sets the kind parameter to "reg" to include a regression line in the plot. This action visualizes the relationship between the 'CRIM' (crime rate) and 'MEDV' (median value of owner-occupied homes) variables, providing insights into their potential correlation and the fitted regression line.

In [58]:
```
sns.jointplot(data=df2, x="CRIM", y="MEDV", kind="reg")
```

Out[58]: <seaborn.axisgrid.JointGrid at 0x1f7ca637310>

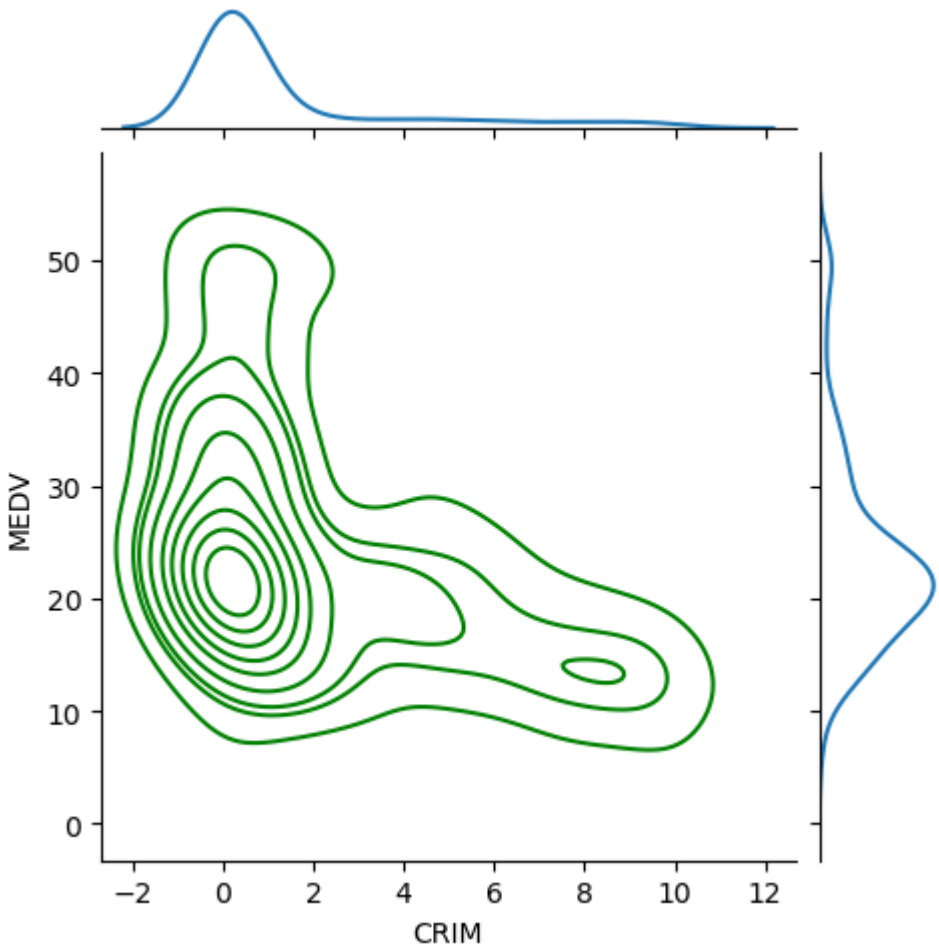



# Join Plots with Seaborn

# Join plot with CRIM and MEDV

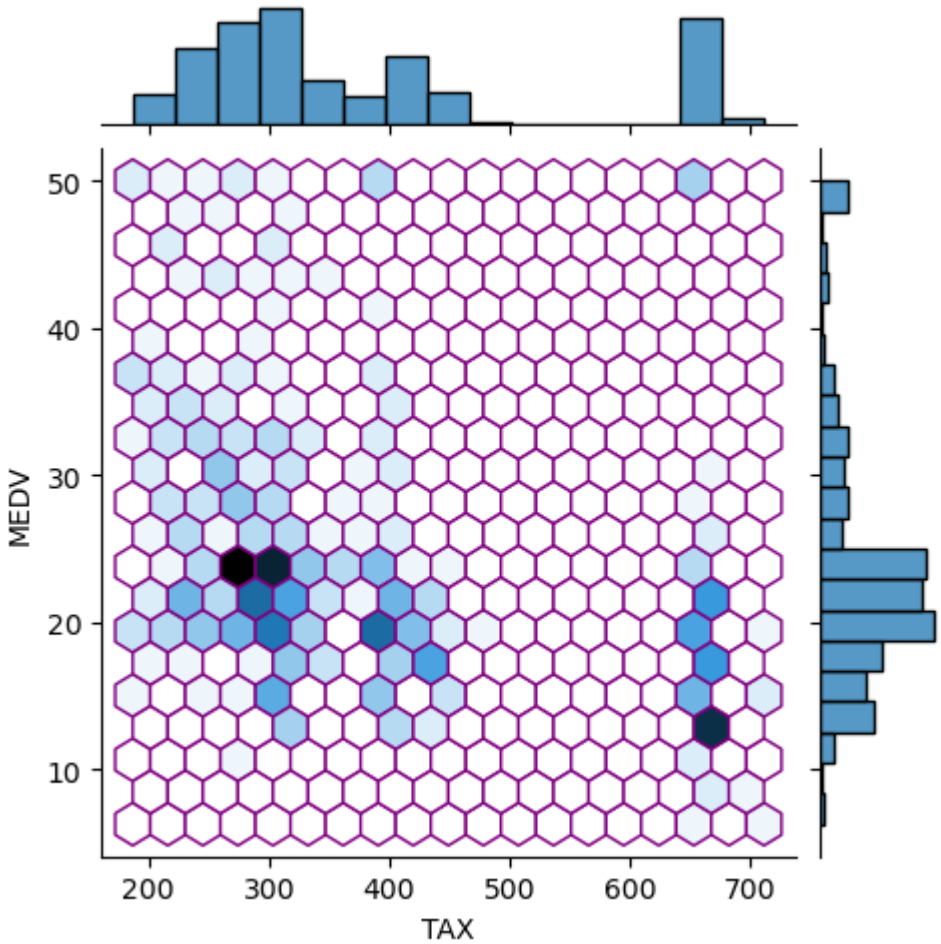
#The code is creating a joint plot using the Seaborn library (sns.jointplot() function) for the variables 'CRIM' and 'MEDV' from the DataFrame df2. It specifies the variable to be plotted on the x-axis (x='CRIM') and the variable to be plotted on the y-axis (y='MEDV'). It also sets the kind parameter to 'kde' to create a joint plot with a kernel density estimation. The height parameter determines the height of the plot, and the joint_kws parameter is used to specify additional styling options, such as the color of the plot. Finally, plt.show() is used to display the joint plot. This action visualizes the relationship between the 'CRIM' (crime rate) and 'MEDV' (median value of owner-occupied homes) variables using a kernel density estimation, allowing for an analysis of their distribution and potential correlation.

In [59]:
```
sns.jointplot(x = 'CRIM', y = 'MEDV', data = df2, kind = 'kde', height = 5
plt.show()
```




The code below is creating a joint plot using the Seaborn library (sns.jointplot() function) for the variables 'TAX' and 'MEDV' from the DataFrame df2. It specifies the variable to be plotted on the x-axis (x='TAX') and the variable to be plotted on the y-axis (y='MEDV'). It sets the kind parameter to 'hex' to create a joint plot with hexagonal binning. The height parameter determines the height of the plot, and the joint_kws parameter is used to specify additional styling options, such as the color of the plot. Finally, plt.show() is used to display the joint plot. This action visualizes the relationship between the 'TAX' (full-value property-tax rate) and 'MEDV' (median value of owner-occupied homes) variables using hexagonal binning, providing insights into their distribution and potential correlation.

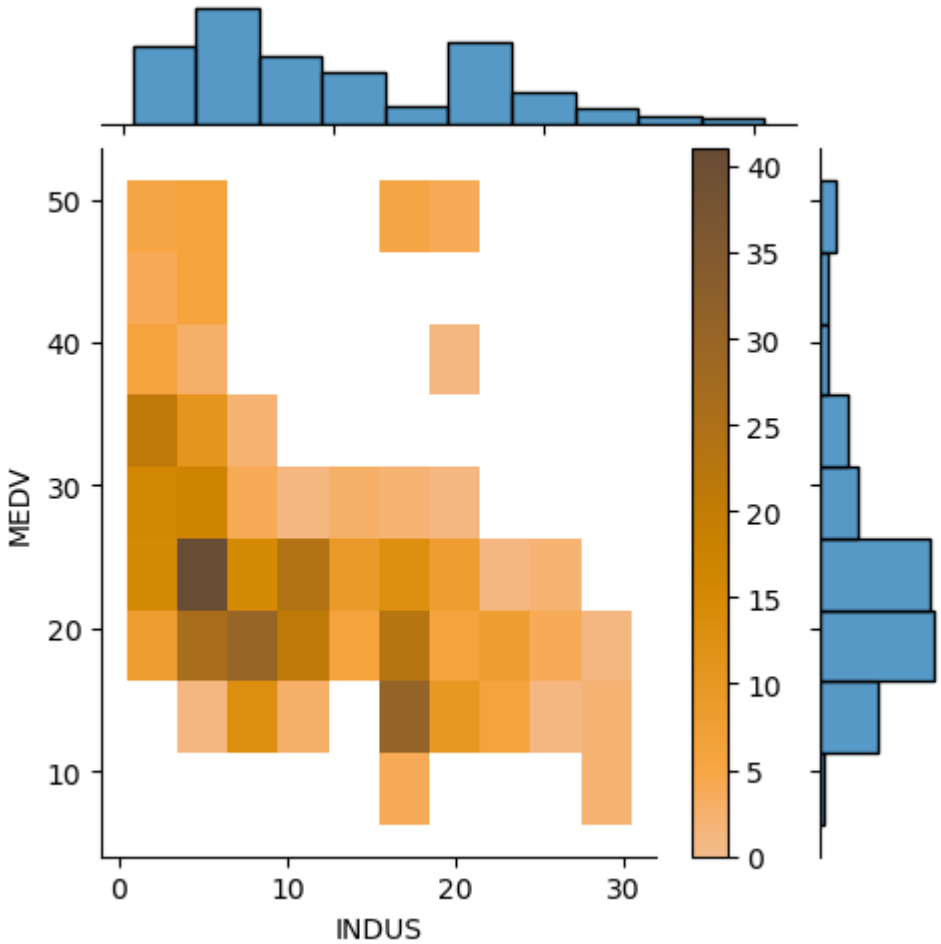
In [61]:
```
sns.jointplot(x = 'TAX', y = 'MEDV', data = df2, kind = 'hex', height = 5,
plt.show()
```




# Join plot with TAX and MEDV

The code is creating a joint plot using the Seaborn library (sns.jointplot() function) for the variables 'INDUS' and 'MEDV' from the DataFrame df2. It specifies the variable to be plotted on the x-axis (x='INDUS') and the variable to be plotted on the y-axis (y='MEDV'). It sets the kind parameter to 'hist' to create a joint plot with histograms. The height parameter determines the height of the plot, and the joint_kws parameter is used to specify additional styling options, such as the color of the plot. The binwidth parameter defines the width of the histogram bins. Finally, plt.show() is used to display the joint plot. This action visualizes the relationship between the 'INDUS' (proportion of non-retail business acres per town) and 'MEDV' (median value of owner-occupied homes) variables using histograms, providing insights into their distribution and potential correlation. The cbar=True option adds a color bar to the plot to indicate the frequency of data points in each bin.

In [62]:
```
sns.jointplot(x = 'INDUS', y = 'MEDV', data = df2, kind = 'hist', height =
plt.show()
```




# Combining the join plots

The code below is creating a PairGrid using the Seaborn library (sns.PairGrid() function) for the DataFrame df2 with a specified height of 10. The PairGrid object allows for creating a grid of subplots where each variable is plotted against every other variable.

The g.map_upper() function is used to map a histogram plot (sns.histplot()) to the upper triangle of the grid. It specifies the number of bins (bins=20) and the binwidth (binwidth=3) for the histogram. The cbar=True option adds a color bar to indicate the frequency of data points in each bin.

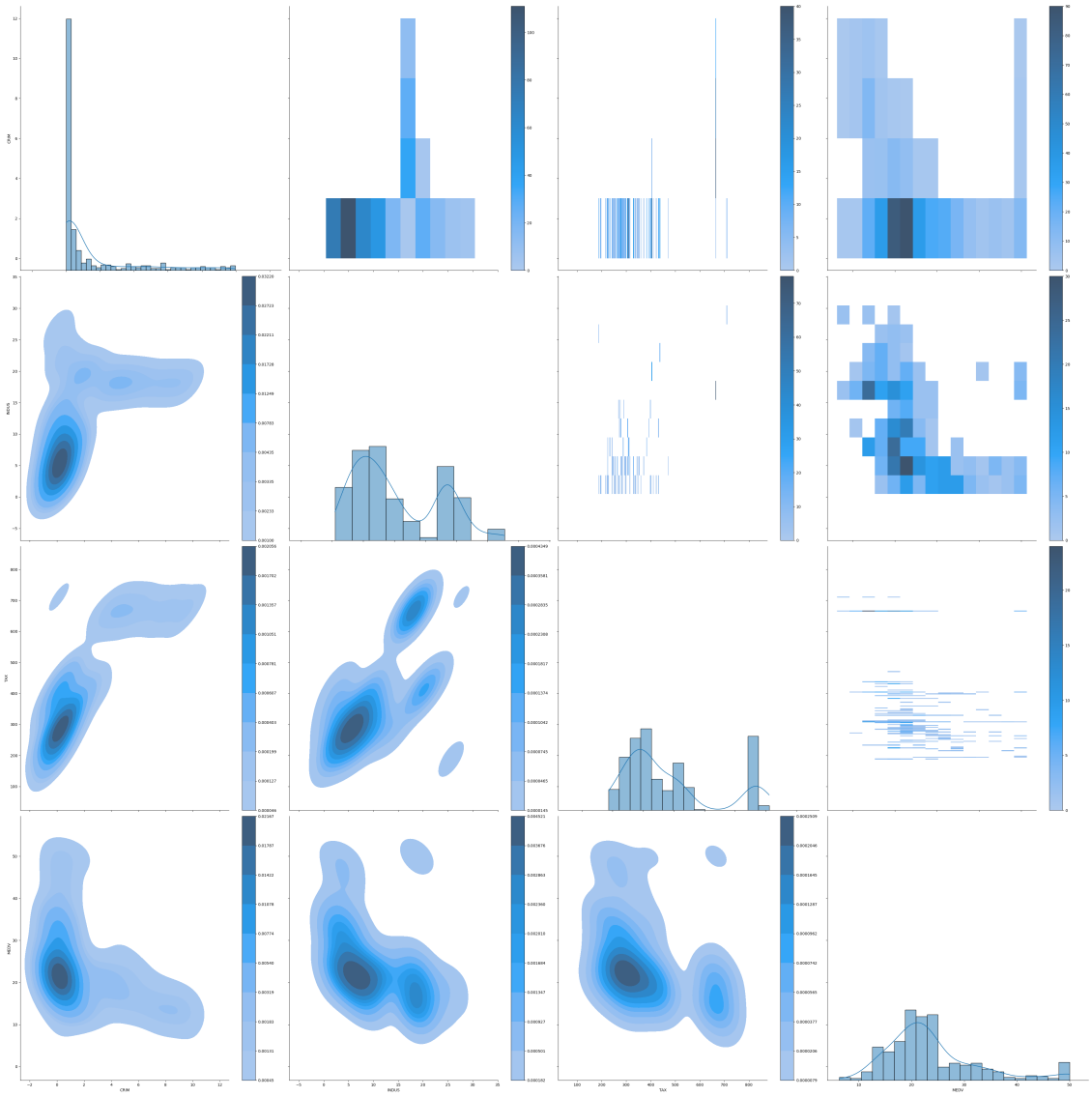
The g.map_lower() function is used to map a kernel density plot (sns.kdeplot()) to the lower triangle of the grid. The fill=True option fills the area under the KDE curve, and cbar=True adds a color bar to indicate the density of data points.

The g.map_diag() function is used to map a histogram plot with KDE (sns.histplot()) to the diagonal of the grid. The kde=True option overlays a KDE plot on top of the histogram, and cbar=True adds a color bar to indicate the density of data points.

Overall, this code generates a grid of subplots containing histograms, KDE plots, and joint plots for each pair of variables in df2, providing a comprehensive visualization of the relationships and distributions between variables.

In [63]:
```
g = sns.PairGrid(df2, height= 10)
g.map_upper(sns.histplot, bins= 20, binwidth=3, cbar=True)
g.map_lower(sns.kdeplot, fill=True, cbar=True)
g.map_diag(sns.histplot, kde=True, cbar=True)
```

Out[63]:
```
<seaborn.axisgrid.PairGrid at 0x1f7ca480130>
```



# Separate Dataset into Input & Output Arrays

The code below is performing the following actions: It stores the values of the DataFrame df2 into a NumPy array using the values attribute. This creates a new variable named array which contains the values of df2. It separates the array into two components: the input (X) and the output (Y) variables. For the input (X), it selects all the rows and the columns from index 0 to 2 (columns CRIM, INDUS, and TAX). For the output (Y), it selects all the rows in the last column (column MEDV). The resulting X and Y are now independent and dependent variables, respectively, for further analysis or modeling tasks.

In [64]:
```
# Store the dataframe values into a numPy array

array = df2.values

# For X (input) [:,3] --> All the rows and columns from 0 up to 3

X = array [:, 0:3]

# For Y (output) [:3] --> All the rows in the last column (MEDV)

Y = array [:,3]
```

# Split into Training/Testing Datasets and Input/Output Arrays

#It sets test_size to 0.33 and indicates that the size of the test sub-dataset is about 33% of the total dataset. It sets the value of the random seed to 7 by seed = 7. This ensures that the random separation of the dataset into training and test subsets is consistent across different runs.Using the train_test_split function from scikit-learn, it splits the input (X) and output (Y) datasets into training and test datasets. 67% of the data will be in the training sub-dataset, and 33% will be in the test sub-dataset. To ensure the split can be replicated, the random_state parameter is set to the value of the seed. The split datasets are then used to train and test models by being assigned to the variables X_train, X_test, Y_train, and Y_test.

In [65]:
```
# Split the dataset --> training sub-dataset:  67%, and test sub-dataset:

test_size = 0.33

# Selection of records to inclue in which sub-dataset must be done randoml

seed = 7

# Split the dataset (both input & output) into training/testing datasets

X_train, X_test, Y_train, Y_test= train_test_split(X,Y, test_size=0.2, ran
```

# The code below is performing the following actions: It builds a DecisionTreeRegressor model by initializing an instance of the DecisionTreeRegressor class with the specified parameters. The random_state parameter is set to the value of the seed, ensuring reproducibility of the model.

It trains the model using the training sub-dataset (X_train and Y_train) by calling the fit method of the model object. The model learns the patterns and relationships in the training data to make predictions on new, unseen data.

The output shown is the representation of the created DecisionTreeRegressor model with its

In [81]:
```
# Build the model

model = DecisionTreeRegressor(random_state=seed)

# Train the model using the training sub-dataset

model.fit(X_train,Y_train)

# Non-Linear --> NO coefficients and the intercept

model = DecisionTreeRegressor(criterion='mse', max_depth=None, max_feature
                              max_leaf_nodes=None, min_samples_leaf=2,
                              min_samples_split=2, min_weight_fraction_lea
                              random_state=seed,
                              splitter='best')
```

# Calculating R-Squared

#The code calculates the R-square value, which is a measure of how well the regression model fits the data. It tests the performance of the model by comparing the predicted values (model.predict(X_test)) with the actual values (Y_test) and returns the R-squared value and then prints the result as "R-Squared." =." " followed by the calculated value.

In [69]:
```
R_squared = model.score(X_test, Y_test)
print('R-Squared = ', R_squared)
```

```
R-Squared =  0.24948500553354236
```

#The code is using the trained model (model) to make predictions on new input data ([[12, 10, 450]]). It predicts the target variable value based on the given input features (12, 10, and 450). The predicted value is returned by the predict method of the model.

In [70]:
```
model.predict([[12,10,450]])
```

Out[70]:
```
array([12.6])
```

#The code below is using the trained model (model) to make predictions on new input data ([[2, 30, 50]]). It predicts the target variable value based on the given input features (2, 30, and 50). The predicted value is returned by the predict method of the model.

In [73]:
```
model.predict([[2,30,50]])
```

Out[73]:
```
array([15.7])
```

# Using K-Fold Cross-Validation, evaluate and validate the algorithm or model.

#The code is evaluating the performance of the algorithm using K-fold cross-validation. It specifies the number of folds (num_folds) and fixes the random seed for reproducibility. The data set is split into folds, and the model is trained and evaluated using negative mean squared error (MSE) as the scoring metric. The results are then averaged across all the folds to obtain the average of the evaluation scores. The average score is printed as the result of the evaluation.

In [74]:
```
# 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'

# Train the model and run K-foLd cross-validation to validate/evaluate the

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 cross va

print("Average of all results from the K-fold Cross Validation, using nega
```

```
Average of all results from the K-fold Cross Validation, using negative m
ean squared error: -76.82251835748792
```