Big Data Lab – Week 8: Data Analysis I

In this lab you will learn and practice data analysis using kNN，Decision Tree and Simple Linear Regression in Python with Scikit-learn.

Scikit-learn (https://scikit-learn.org/stable/) provides simple and efficient machine learning tools for data mining and data analysis in Python. It is a free, open-source library built on NumPy, SciPy and matplotlib.

Jupyter Notebook is used in this document, while you can use Spyder, Jupyter Notebook, Google Colab or any other python development environments you prefer in the lab.

As mentioned in week 7 lab, if you are using the VM, do not forget to run the following command in a terminal window to active Python 3 before you start a python tool:

```
source activate py37
```

If you are using Google Colab, save the data files in your google drive. Do not forget to mount your google drive and change working directory. You can refer to "*Mount Google drive and change working directory in Colab.pdf*" in week 7 folder on GCUlearn.

Task 1: Build a kNN model

KNN (K-Nearest Neighbour) is a simple supervised classification algorithm we can use to assign a class to new instance. KNN does not make any assumptions on the data distribution. It keeps all the training data to make future predictions by computing the similarity between an input sample and each training instance.

1. Download the data file "***iris.csv***" from GCU-Learn
2. Import the *Iris* dataset and check the features

```
import pandas as pd

df = pd.read_csv("iris.csv")
print(df.shape)

print(df.head(5))
```

Run this script, you will see output like below:

```
(150, 5)
```

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

So there are 150 rows, 5 columns in the *iris* dataset.

3. Similar to week 7 lab, exploring and pre-processing the iris data

```
# Chaecking the types of data
print(df.dtypes)
```

```
sepal length    float64
sepal width     float64
petal length    float64
petal width     float64
class            object
dtype: object
```

```
# Checking rows containing duplicate data
duplicate_rows_df = df[df.duplicated()]
print("number of duplicate rows: ", duplicate_rows_df.shape)
```

```
number of duplicate rows:  (3, 5)
```

```
# Dropping the duplicates
df = df.drop_duplicates()
# Counting the number of rows after removing duplicates
df.count()
```

```
sepal length    147
sepal width     147
petal length    147
petal width     147
class           147
dtype: int64
```

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

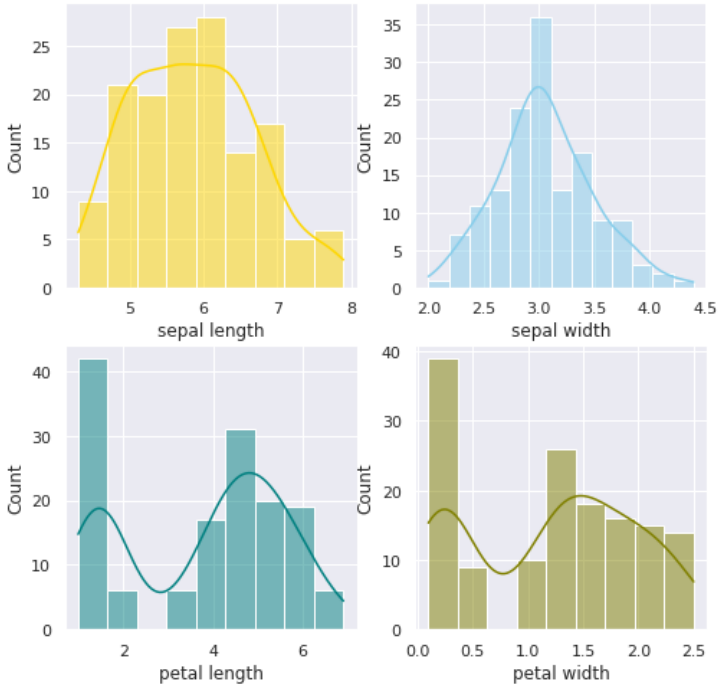
```
sepal length    0
sepal width     0
petal length    0
petal width     0
class           0
dtype: int64
```

```
# Printing summary statistics on attributes
print(df.describe())
```

```
       sepal length  sepal width  petal length  petal width
count    147.000000   147.000000    147.000000   147.000000
mean       5.856463     3.055782      3.780272     1.208844
std        0.829100     0.437009      1.759111     0.757874
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.400000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
```

Plot histograms of numeric variables: (optional)

```
# import libraries
import seaborn as sns
import matplotlib.pyplot as plt

# set a grey background (use sns.set_theme() if seaborn version 0.11.0 or above)
sns.set(style="darkgrid")

fig, axs = plt.subplots(2, 2, figsize=(8, 8))

# plot the histograms
sns.histplot(data=df, x="sepal length", kde=True, color="gold", ax=axs[0, 0])
sns.histplot(data=df, x="sepal width", kde=True, color="skyblue", ax=axs[0, 1])
sns.histplot(data=df, x="petal length", kde=True, color="teal", ax=axs[1, 0])
sns.histplot(data=df, x="petal width", kde=True, color="olive", ax=axs[1, 1])

plt.show()
```



This code cell works fine with Google Colab. If you are using Anaconda based Spyder or Jupyter Notebook, you may get an error message like:

AttributeError: Module 'seaborn' has no attribute 'histplot'

This is because that the '*histplot'* is only recently available starting in ***seaborn 0.11.0***, and the seaborn in anaconda3 is version 0.10.0. To get this sorted, you can install the new seaborn by running the following pip command in an Anaconda Prompt terminal:

pip install seaborn==0.11.0

After this installation, restart your Spyder or Jupyter Notebook, then the code should be run without error messages.

4. Specify the input variables and the target variable

   We will use the first four columns, i.e., sepal length, sepal width, petal length and petal width, as ***input variables***, and the last column, i.e., class, as ***target variable*** to build a kNN classifier.

```
#create a dataframe with all variables except the class column
X = df.drop(['class'], axis=1)

#check that the class variable has been removed
X.head()
```

   Run the script, you will see:

	sepal length	sepal width	petal length	petal width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

```
#separate target values
y = df["class"].values

#view target values
print(y[0:5])
```

   Run the script, you will see:

```
['Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa']
```

5. Split the dataset into two pieces: training set and testing set, using the '***train_test_split***' function from sklearn. The testing data will be used to check the accuracy of the kNN model.

```
# using the 'train_test_split' function from sklearn to split the dataset
from sklearn.model_selection import train_test_split

X_train, X_Vali_test, y_train, y_Vali_test = train_test_split(X, y,
                                    test_size=0.2, random_state=1, stratify=y)
```

Check the size of the input training set:

```
print(X_train.shape)
```

Because the "***test_size***" in '*train_test_split*' is set as **0.2**, which means **20%** of the data is used as testing data, and the other 80% is training data. There are 147 rows in the *iris* dataset, so you should see the size of the input training set as:

```
(117, 4)
```

6. Build the kNN classifier

We will create a new k-NN classifier and set '***n_neighbors***' as '**3**', which means a new data point is labelled with by majority from the 3 nearest points.

Next we will use the '***fit***' function and pass in our training data as parameters to fit the kNN model to the training data.

```
from sklearn.neighbors import KNeighborsClassifier

# Create KNN classifier
knn = KNeighborsClassifier(n_neighbors = 3)

# Fit the classifier to the data
knn.fit(X_train,y_train)
```

7. Testing the model

Once the model is trained, we can use the '***predict'*** function on our model to make predictions on our testing data. To save space, we will only show the first 5 predictions of our testing set.

```
#show first 5 model predictions on the test data
print(knn.predict(X_test)[0:5])
```

You may see output similar to below:

```
['Iris-virginica' 'Iris-setosa' 'Iris-versicolor' 'Iris-setosa'
 'Iris-setosa']
```

Now let's see how accurate our model is on the full testing set. To do this, we will use the '***score'*** function and pass in the testing input and target data to see how well our model predictions match up to the actual results.

```
#check accuracy of our model on the test data
knn.score(X_Vali_test, y_Vali_test)
```

Run this script and we got:

```
0.9666666666666667
```

Which means, our kNN classifier has an accuracy of approximately 96.67%.

8. Tune the model

   In general, the accuracy rises as the model complexity increases. For kNN the model complexity is determined by the value of K. Larger K value leads to smoother decision boundary (less complex model). Smaller K leads to more complex model (may lead to overfitting). Accuracy penalizes models that are too complex (over fitting) or not complex enough (underfit).

   Change the value of K, i.e., the value of '*n_neighbours*' in step 5, for example, set k as 2, 10, 20, etc. Check how the accuracy changes accordingly.

   You can also change the distance metric and observe how this will affect the accuracy.

9. More data pre-processing

   You can try to scale the variables into the range, e.g., [0,1], and check how this will affect the accuracy.

Task 2: Construct a Decision Tree classifier

A decision tree is a flowchart-like tree structure. The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in recursively manner. This flowchart-like structure helps you in decision making. It's visualization like a flowchart diagram which easily mimics the human level thinking. That is why decision trees are easy to understand and interpret.

Decision Tree is a white box type of ML algorithm. It shares internal decision-making logic, which is not available in the black box type of algorithms such as Neural Network. The decision tree is a distribution-free or non-parametric method, which does not depend upon probability distribution assumptions.

1. Download the data file "***diabetes_with_head.csv***" from GCU-Learn

2. Import the required libraries

```
# load libraries

import numpy as np
# import pandas
import pandas as pd
# Import Decision Tree Classifier
from sklearn.tree import DecisionTreeClassifier
# Import train_test_split function
from sklearn.model_selection import train_test_split
#Import scikit-learn metrics module for accuracy calculation
from sklearn import metrics
```

3. Import the *diabetes_with_head* dataset and check the features

```
# loading data

df2 = pd.read_csv("diabetes_with_head.csv")
print(df2.shape)

df2.head()
```

Run this script, you will see output like below:

```
(768, 9)
```

	pregnant	glucose	bp	skin	insulin	BMI	pedigree	Age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

So there are 768 rows, 9 columns in the *diabetes_with_head* dataset.

4. Similar to week 7 lab, exploring and pre-processing the diabetes data

```
# check the type of data
print(df2.dtypes)
```

```
pregnant      int64
glucose       int64
bp            int64
skin          int64
insulin       int64
BMI         float64
pedigree    float64
age           int64
label         int64
dtype: object
```

```
# check duplicate rows
duplicate_rows = df2[df2.duplicated()]
print("number of duplicate rows: ", duplicate_rows.shape)
```

```
number of duplicate rows:  (0, 9)
```

There are no duplicated rows, so no further process needed.

```
# print summary statictics
print(df2.describe())
```

```
          pregnant     glucose          bp  ...    pedigree         age       label
count  768.000000  768.000000  768.000000  ...  768.000000  768.000000  768.000000
mean     3.845052  120.894531   69.105469  ...    0.471876   33.240885    0.348958
std      3.369578   31.972618   19.355807  ...    0.331329   11.760232    0.476951
min      0.000000    0.000000    0.000000  ...    0.078000   21.000000    0.000000
25%      1.000000   99.000000   62.000000  ...    0.243750   24.000000    0.000000
50%      3.000000  117.000000   72.000000  ...    0.372500   29.000000    0.000000
75%      6.000000  140.250000   80.000000  ...    0.626250   41.000000    1.000000
max     17.000000  199.000000  122.000000  ...    2.420000   81.000000    1.000000

[8 rows x 9 columns]
```

```
# dispaly the first 10 rows
print(df2.head(10))
```

```
   pregnant  glucose  bp  skin  insulin   BMI  pedigree  age  label
0         6      148  72    35        0  33.6     0.627   50      1
1         1       85  66    29        0  26.6     0.351   31      0
2         8      183  64     0        0  23.3     0.672   32      1
3         1       89  66    23       94  28.1     0.167   21      0
4         0      137  40    35      168  43.1     2.288   33      1
5         5      116  74     0        0  25.6     0.201   30      0
6         3       78  50    32       88  31.0     0.248   26      1
7        10      115   0     0        0  35.3     0.134   29      0
8         2      197  70    45      543  30.5     0.158   53      1
9         8      125  96     0        0   0.0     0.232   54      1
```

```
# Count the number of null values in each column
print(df2.isnull().sum())
```

```
pregnant    0
glucose     0
bp          0
skin        0
insulin     0
BMI         0
pedigree    0
age         0
label       0
dtype: int64
```

As mentioned in week 7 lab, on some columns, a value of zero does not make sense and indicates an invalid or missing value. So we should treat zero values in those columns (i.e., "glucose","bp","skin","insulin","BMI" in this diabetes data) as invalid/ missing values.

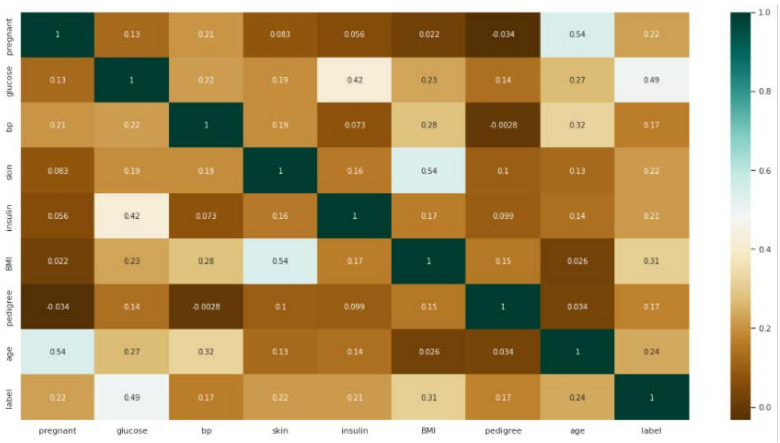
In this case, missing values are imputed with mean of the variable/column values.

```
# count of the number of missing values on each of these columns
print((df2[["glucose","bp","skin","insulin","BMI"]] == 0).sum())

# mark zero values as missing (with the value of NaN)
df2[["glucose","bp","skin","insulin","BMI"]] = df2[["glucose",
                        "bp","skin","insulin","BMI"]].replace(0, np.NaN)
# check the number of NaN values in each column
print(df2.isnull().sum())
# print the first 10 rows of data
print(df2.head(10))

# fill missing values with mean column values
df2.fillna(df2.mean(), inplace=True)
# check if there is still any NaN values in the dataset
print(df2.isnull().sum())
# check the imputated the first 10 rows of data
print(df2.head(10))
```

(output of this code cell is too long, so not included)

```
# Finding the correlations between the variables.
plt.figure(figsize=(20,10))
corl= df2.corr()
sns.heatmap(corl,cmap="BrBG",annot=True)
print(corl)
```

```
            pregnant   glucose        bp  ...  pedigree       age     label
pregnant    1.000000  0.127911  0.208522  ... -0.033523  0.544341  0.221898
glucose     0.127911  1.000000  0.218367  ...  0.137060  0.266534  0.492928
bp          0.208522  0.218367  1.000000  ... -0.002763  0.324595  0.166074
skin        0.082989  0.192991  0.192816  ...  0.100966  0.127872  0.215299
insulin     0.056027  0.420157  0.072517  ...  0.098634  0.136734  0.214411
BMI         0.021565  0.230941  0.281268  ...  0.153400  0.025519  0.311924
pedigree   -0.033523  0.137060 -0.002763  ...  1.000000  0.033561  0.173844
age         0.544341  0.266534  0.324595  ...  0.033561  1.000000  0.238356
label       0.221898  0.492928  0.166074  ...  0.173844  0.238356  1.000000

[9 rows x 9 columns]
```



The maximum correlation coefficient in this matrix is 0.54, which means variables in this data are not significantly correlated, so no further processes.

5. Specify the input variables and the target variable.

   We will use the first eight columns as input variable and the last column, i.e., label, as target variable to build a Decision Tree.

```
#split dataset into input variable X and target variable y

X = df2.drop(['label'], axis=1)

#check that the class variable has been removed
X.head()
```

Run the script, you will see:

	pregnant	glucose	bp	skin	insulin	BMI	pedigree	Age
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33

```
#separate target variable
y = df2.label

#view target values
print(y[0:5])
```

Run the script, you will see:

```
0    1
1    0
2    1
3    0
4    1
Name: label, dtype: int64
```

6. Split the dataset into two pieces: 70% as training set and 30% as testing set.

```
# Split dataset into training set and test set:  70% training and 30% testing
X_train, X_Vali_test, y_train, y_Vali_test = train_test_split(X, y,
                                                 test_size=0.3, random_state=1)

print(X_train.shape)
```

The size of the input training set is

```
(537, 8)
```

7. Construct the Decision Tree. Use the 'fit' function and pass in our training data as parameters to train the Decision Tree classifier.

```
# Create Decision Tree classifer object
clf = DecisionTreeClassifier()

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)
```

8. Testing the accuracy of the model

Once the model is trained, we can use the '*predict'* function on our model to make predictions on our test data and calculate the accuracy score.

```
#Predict the response for test dataset
y_pred = clf.predict(X_Vali_test)

# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_Vali_test, y_pred))
```

You may get an accuracy similar to below:

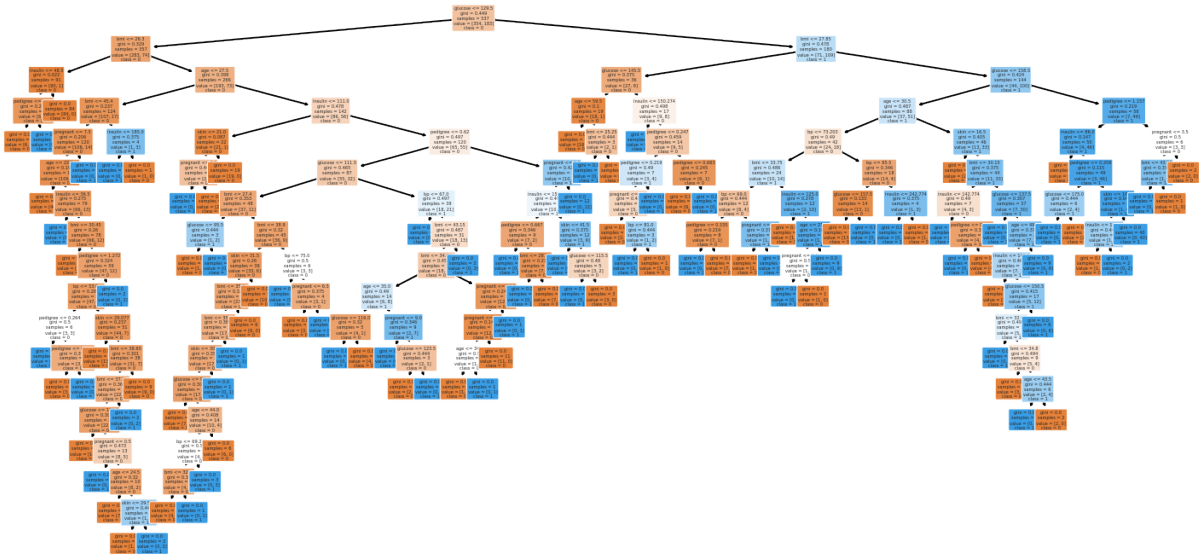
```
Accuracy: 0.7489177489177489
```

Which means, our Decision Tree classifier has an accuracy of approximately 74.89%.

9. Visualising the Decision Tree (optional)

```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

feature_cols = ['pregnant', 'glucose','bp','skin',
                'insulin', 'bmi','pedigree', 'age']

plt.figure(figsize=(25,12))
a = plot_tree(clf,
              feature_names=feature_cols,
              class_names=['0','1'],
              filled=True,
              rounded=True,
              fontsize = 5)
for o in a:
    arrow = o.arrow_patch
    if arrow is not None:
        arrow.set_edgecolor('black')
        arrow.set_linewidth(2)
```

Run the script and you will see the decision tree as below:



10. Tune the decision tree model

    Let us limit the max-depth of the Decision Tree as ‘3’ and use “entropy” as the criterion to measure the quality of a split.

```
# Create Decision Tree classifer object
clf_02 = DecisionTreeClassifier(criterion="entropy", max_depth=3)

# Train Decision Tree Classifer
clf_02 = clf_02.fit(X_train,y_train)

#Predict the response for test dataset
y_pred = clf_02.predict(X_Vali_test)

# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_Vali_test, y_pred))
```

    Train the updated decision tree and test it. You may get the accuracy as below:

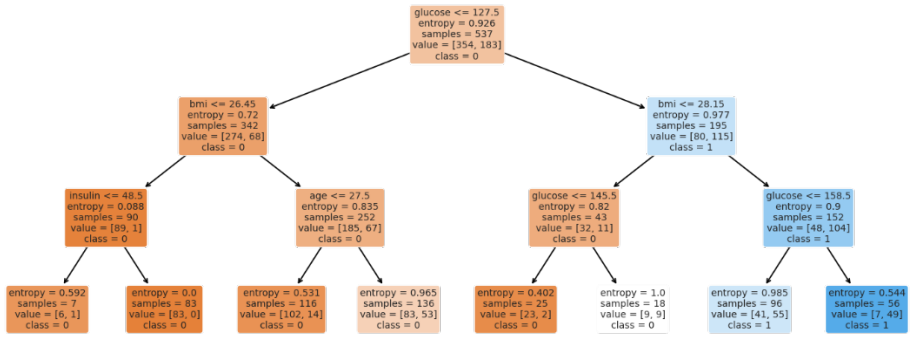
```
Accuracy: 0.7662337662337663
```

    This demonstrated that the updated Decision Tree classifier has better accuracy of approximately 76.62%.

11. Visualized the tuned decision tree (optional)

```
plt.figure(figsize=(25,10))
a = plot_tree(clf_02,
              feature_names=feature_cols,
              class_names=['0','1'],
              filled=True,
              rounded=True,
              fontsize = 14)
for o in a:
    arrow = o.arrow_patch
    if arrow is not None:
        arrow.set_edgecolor('black')
        arrow.set_linewidth(2)
```

Run this script, you will see the tuned decision tree as below:



Task 3: Simple linear regression

In this task, you will practice how to implement Simple Linear Regression in Python. Again Jupyter Notebook is used to present this task, while you can choose whatever python IDE you like.

1. Download data file '***insurance.csv'*** from GCULearn.

2. Load the '*insurance.csv* dataset into Python as a Pandas DataFrame; and print summary statistics on each attribute

```
from pandas import read_csv
import matplotlib.pyplot as plt

# load the dataset
dataset = read_csv('insurance.csv', header=None)

# print the summary statistics on each attributes
print(dataset.describe())
```

Below is the output you will have:

```
                0           1
count   63.000000   63.000000
mean    22.904762   98.187302
std     23.351946   87.327553
min      0.000000    0.000000
25%      7.500000   38.850000
50%     14.000000   73.400000
75%     29.000000  140.000000
max    124.000000  422.200000
```

3. Check the duplication

```
# Rows containing duplicate data
duplicate_rows_df = df[df.duplicated()]
print("number of duplicate rows: ", duplicate_rows_df.shape)
```

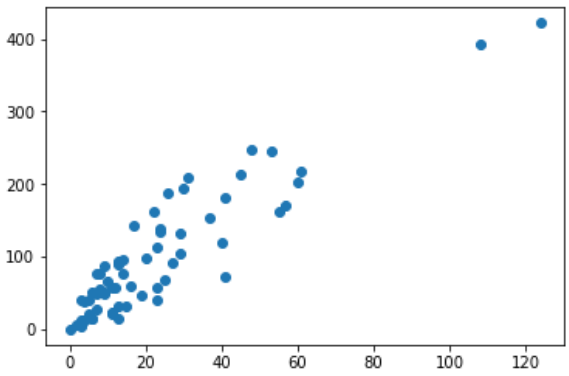
```
number of duplicate rows:  (0, 5)
```

No duplication in this data, no further processes.

4. Specify the input variable (x) and the target variable (y), and plot the data

```
# specify the input variable (x) and the target variable (y)
x = dataset[0]
y = dataset[1]

# plot the data
# the following line is not needed in Spyder
%matplotlib inline
plt.scatter(x,y)
```

You should get a scatter plot like below:



5. In simple linear regression, the equation for the best fitting regression line is:

$$\tilde{y}_i = b_0 + b_1 x_i \quad (1)$$

where

$$b_0 = \bar{y} - b_1 \bar{x}$$

$$b_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

In order to estimate b_0 and b_1, we can define the following functions:

```
# Calculate the mean value of a list of numbers
def mean(values):
    return sum(values) / float(len(values))

# Calculate the variance of a list of numbers
def variance(values, mean):
    return sum([(s - mean)**2 for s in values])

# Calculate covariance between x and y
def covariance(s, mean_s, t, mean_t):
    covar = 0.0
    for i in range(len(s)):
        covar += (s[i] - mean_s) * (t[i] - mean_t)
    return covar
```

So coefficients in the best regression line can be estimated using:

```
# Function to calculate coefficients
def coefficients(in_var, target):
    x_mean, y_mean = mean(in_var), mean(target)
    b1 = covariance(in_var, x_mean, target, y_mean) / variance(in_var, x_mean)
    b0 = y_mean - b1 * x_mean
    return [b0, b1]
```

6. Using the defined '*coefficients*' function to estimate the best regression in between the two variables in the '*insurance*' dataset.

```
# estimate the best fitting regression line
b0, b1 = coefficients(x, y)

print('Coefficients: b0=%.3f, b1=%.3f' % (b0, b1))
print('The best regrssion line is: y = %.2f + %.2f * x' % (b0, b1))
```

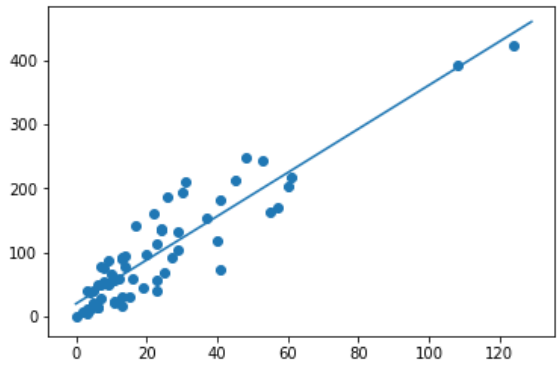
You will see the value of coefficients and the best regression line function as:

```
Coefficients: b0=19.994, b1=3.414
The best regrssion line is: y = 19.99 + 3.41 * x
```

7. Plot the best fitting line

```
# in irder to plot the regression line, calculate points on the line
x1 = range(130)
y1 = []
for i in x1:
    #x1.append(i)
    y1.append(b0 + b1*i)

# plot the regression line
plt.plot(x1,y1)
# plot the dataset in the same figure
plt.scatter(x,y)
```

You should get a figure like below:



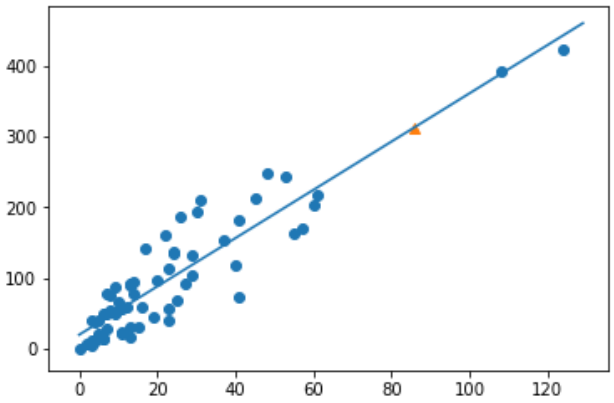
8. Once the coefficients are estimated, you can use them to make predictions (as shown in equation 1 above).

   For example, assuming in a new instance, the input variable is 85.72, and you can predict the value of corresponding target variable:

```
# assuming in a new instance, the input variable is 85.72
x_k = 85.72

# predict the value of target varible using the regression model
y_k = b0 + b1 * x_k
print('the new instacnce is: %.2f, %.2f' % (x_k, y_k))

# plot the regression line and other instances
plt.plot(x1,y1)
plt.scatter(x,y)

# plot the new instance in the figure as a triangle
plt.scatter(x_k, y_k, marker='^')
```

   Below is the output. The new instance is plotted as a triangle in the figure.

```
the new instacnce is: 85.72, 312.63
<matplotlib.collections.PathCollection at 0x7fc250d0b410>
```



9. Try to predict the target variable when the input variable has different value, e.g. 42.16, 65.3, etc.