Linear Regression with Python Scikit Learn¶ In this section we will see how the Python Scikit-Learn library for machine learning can be used to implement regression functions. We will start with simple linear regression involving two variables.

Simple Linear Regression In this regression task we will predict the percentage of marks that a student is expected to score based upon the number of hours they studied. This is a simple linear regression task as it involves just two variables

In [1]:

```
# Importing all libraries required in this notebook
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:

```
# Reading data from remote link
url = "http://bit.ly/w-data"
s_data = pd.read_csv(url)
print("Data imported successfully")

s_data.head(10)
```

```
Data imported successfully
```

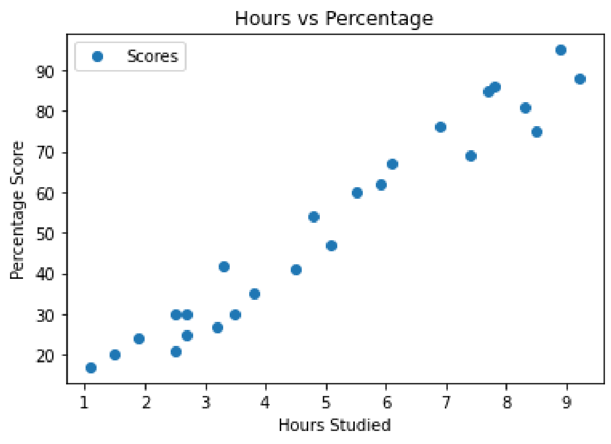
Out[2]:

	Hours	Scores
0	2.5	21
1	5.1	47
2	3.2	27
3	8.5	75
4	3.5	30
5	1.5	20
6	9.2	88
7	5.5	60
8	8.3	81
9	2.7	25

Let's plot our data points on 2-D graph to eyeball our dataset and see if we can manually find any relationship between the data. We can create the plot with the following script:

In [3]:

```
# Plotting the distribution of scores
s_data.plot(x='Hours', y='Scores', style='o')
plt.title('Hours vs Percentage')
plt.xlabel('Hours Studied')
plt.ylabel('Percentage Score')
plt.show()
```



From the graph above, we can clearly see that there is a positive linear relation between the number of hours studied and percentage of score.

Preparing the data

The next step is to divide the data into "attributes" (inputs) and "labels" (outputs)

In [4]:

```
X = s_data.iloc[:, :-1].values
y = s_data.iloc[:, 1].values
```

Now that we have our attributes and labels, the next step is to split this data into training and test sets. We'll do this by using Scikit-Learn's built-in train_test_split() method:

In [5]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                            test_size=0.2, random_state=0)
```

Training the Algorithm¶

We have split our data into training and testing sets, and now is finally the time to train our algorithm.

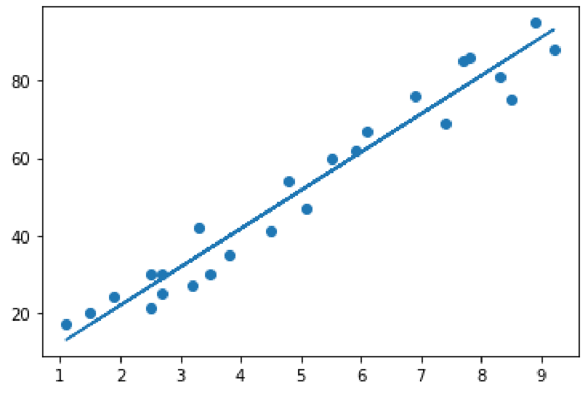
In [7]:

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)

print("Training complete.")
```

```
Training complete.
```

In [8]:

```
# Plotting the regression line
line = regressor.coef_*X+regressor.intercept_

# Plotting for the test data
plt.scatter(X, y)
plt.plot(X, line);
plt.show()
```



Making Predictions¶

Now that we have trained our algorithm, it's time to make some predictions.

In [9]:

```
print(X_test) # Testing data - In Hours
y_pred = regressor.predict(X_test) # Predicting the scores
```

```
[[1.5]
 [3.2]
 [7.4]
 [2.5]
 [5.9]]
```

In [10]:

```
# Comparing Actual vs Predicted
df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
df
```

Out[10]:

	Actual	Predicted
0	20	16.884145
1	27	33.732261
2	69	75.357018
3	30	26.794801
4	62	60.491033

In [13]:

```
# You can also test with your own data
hours = 9.25

own_pred = regressor.predict(hours)
print("No of Hours = {}".format(hours))
print("Predicted Score = {}".format(own_pred[0]))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-afaa235fd97f> in <module>
      2 hours = 9.25
      3 
----> 4 own_pred = regressor.predict(hours)
      5 print("No of Hours = {}".format(hours))
      6 print("Predicted Score = {}".format(own_pred[0]))

~\Anaconda3\lib\site-packages\sklearn\linear_model\_base.py in predict(self, X)
    236             Returns predicted values.
    237         """
--> 238         return self._decision_function(X)
    239 
    240     _preprocess_data = staticmethod(_preprocess_data)

~\Anaconda3\lib\site-packages\sklearn\linear_model\_base.py in _decision_function(self, X)
    218         check_is_fitted(self)
    219 
--> 220         X = check_array(X, accept_sparse=['csr', 'csc', 'coo'])
    221         return safe_sparse_dot(X, self.coef_.T,
    222                                dense_output=True) + self.intercept_

~\Anaconda3\lib\site-packages\sklearn\utils\validation.py in inner_f(*args, **kwargs)
     61             extra_args = len(args) - len(all_args)
     62             if extra_args <= 0:
---> 63                 return f(*args, **kwargs)
     64 
     65             # extra_args > 0

~\Anaconda3\lib\site-packages\sklearn\utils\validation.py in check_array(array, accept_sparse, accept_large_sparse, dtype, order, copy, for
ce_all_finite, ensure_2d, allow_nd, ensure_min_samples, ensure_min_features, estimator)
    632                     "Reshape your data either using array.reshape(-1, 1) if "
    633                     "your data has a single feature or array.reshape(1, -1) "
--> 634                     "if it contains a single sample.".format(array))
    635             # If input is 1D raise error
    636             if array.ndim == 1:

ValueError: Expected 2D array, got scalar array instead:
array=9.25.
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape(1, -1) if it contains a single sample.
```

Evaluating the model

The final step is to evaluate the performance of algorithm. This step is particularly important to compare how well different algorithms perform on a particular dataset. For simplicity here, we have chosen the mean square error. There are many such metrics.

In [12]:

```
from sklearn import metrics
print('Mean Absolute Error:',
      metrics.mean_absolute_error(y_test, y_pred))
```

```
Mean Absolute Error: 4.183859899002975
```