Python Tutorial for Vectorizing form of Implementation

Example 1: Illustrative Example of Neural Network Implementation via Vectorizing the function

In [22]:

```
import numpy as np

# Small number of data samples

# x0, x1, x2 = 1., 2., 3.
# bias, w1, w2 = 0.1, 0.3, 0.5

# x = [x0, x1, x2]
# w = [bias, w1, w2]
# x_vec, w_vec = np.array(x), np.array(w)

# Large number of data samples
x, w = np.random.rand(100000), np.random.rand(100000)
x_vec, w_vec = np.array(x), np.array(w)
```

In [23]:

```
# Python code to demonstrate the working of  # zip()

# # initializing lists
# name = [ "Manjeet", "Nikhil", "Shambhavi", "Astha" ]
# marks = [ 40, 50, 60, 70 ]

# # using zip() to map values
# mapped = zip(name, marks)
# # converting values to print as set
# mapped = set(mapped)

# # printing resultant values
# print ("The zipped result is : ",end="")
# print (mapped)

# #Unzipping the Value Using zip()
# c, v, =  zip(*mapped)
# print('c =', c)
# print('v =',v)
```

In [24]:

```
# Neural network output with For Loop  statement
def forloop(x, w):
    z = 0.
    for i in range(len(x)):
        z += x[i] * w[i]
    return z

# Neural network  output with listcomprehension statement
def listcomprehension(x, w):
    z = sum(x_i*w_i for x_i, w_i in zip(x, w))
    return z

# Neural network output with  Vectorized  form
def vectorized(x, w):
     z = x_vec.dot(w_vec)
    #  z = (x_vec.transpose()).dot(w_vec)
     return z


```

Comparison of Processing Speed of above three different forms of implemenattion

In [25]:

```
# Method-1: forloop
import time
t10 = time.time()
print(forloop(x,w))
t11 = time.time()
time_forloop = t11 - t10
print(time_forloop)
```

```
25005.565200480993
0.03804349899291992
```

In [26]:

```
# Method-2: ListComprehension Implemenattion
import time
t20 = time.time()
print(listcomprehension(x,w))
t21 = time.time()
time_listComp = t21 - t20
print(time_listComp)
```

```
25005.565200480993
0.02210259437561035
```

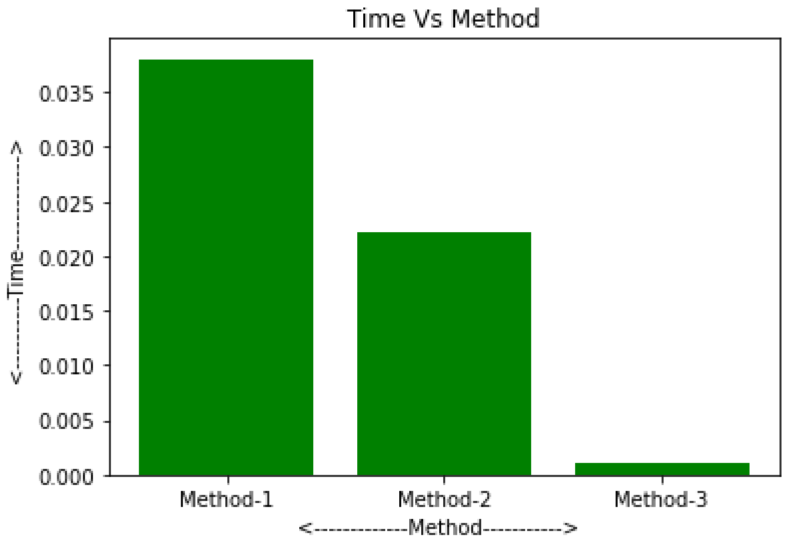
In [27]:

```
# Method-3: Vectorized Implemenattion
import time
t30 = time.time()
print(vectorized(x_vec,w_vec))
t31 = time.time()
time_vectorized = t31 - t30
print(time_vectorized)
```

```
25005.565200480753
0.0009987354278564453
```

In [28]:

```
import matplotlib.pyplot as plt
%matplotlib inline
# plt.style.use('ggplot')

x = ['Method-1', 'Method-2', 'Method-3']
Processing_Times = [time_forloop, time_listComp, time_vectorized]


x_pos = [i for i, _ in enumerate(x)]

plt.bar(x_pos, Processing_Times, color='green')
plt.xlabel("<-------------Method----------->  ")
plt.ylabel("<----------Time-------------> ")
plt.title(" Time Vs Method ")

plt.xticks(x_pos, x)

plt.show()
```



Example 2: Illustrative Example for Predicting House sales price using Boston house dataset

In [29]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.datasets import load_boston
boston_data = load_boston()
print(boston_data['DESCR'])
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of black people by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/ (https://archive.ics.uci.edu/ml/machine-learning-databases/housing/)


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
```

...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

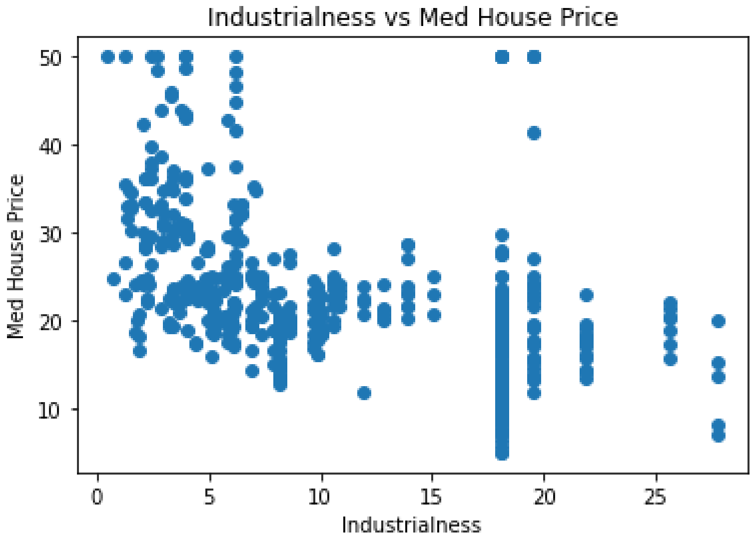
The Boston house-price data has been used in many machine learning papers that address regression
problems.

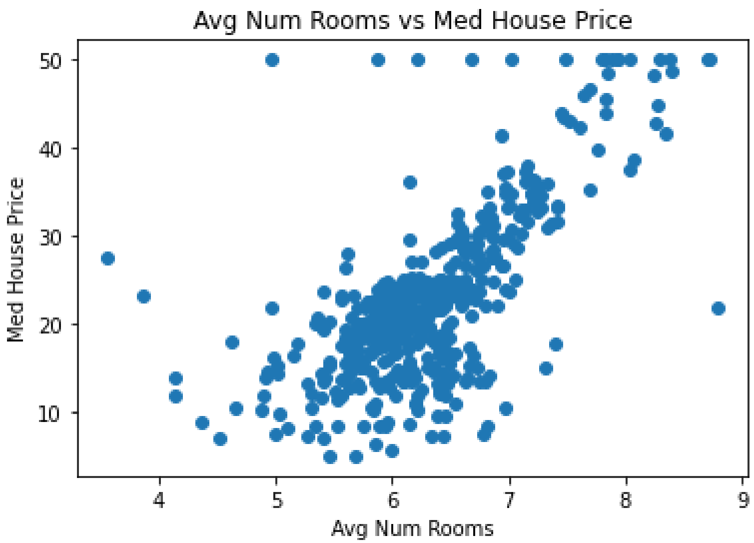
.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.

In [30]:

```python
# take the boston data
data = boston_data['data']
# we will only work with two of the features: INDUS and RM
# x_input = data[:, [2,]]       # for single feature of input data (INDUS)
x_input = data[:, [2,5]]         # for two features of input data (INDUS and RM)
# x_input = data[:, [2,5,7]]         # for three features of input data (INDUS,RM, and DIS)
# x_input = data[:, ]                        # All features of input data
y_target = boston_data['target']
# print(x_input.shape[1])
# print(x_input)
# print(y_target.shape[0])
# print(y_target)

# Individual plots for the two features:
plt.title('Industrialness vs Med House Price')
plt.scatter(x_input[:, 0], y_target)
plt.xlabel('Industrialness')
plt.ylabel('Med House Price')
plt.show()

plt.title('Avg Num Rooms vs Med House Price')
plt.scatter(x_input[:, 1], y_target)
plt.xlabel('Avg Num Rooms')
plt.ylabel('Med House Price')
plt.show()

# plt.title('Avg weighted distances  vs Med House Price')
# plt.scatter(x_input[:, 2], y_target)
# plt.xlabel('Avg weighted distances ')
# plt.ylabel('Med House Price')
# plt.show()

```





Define cost function: Non-vectorized form

$$\mathcal{E}(y, t) = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)})^2$$

$$\mathcal{E}(y,t) = \frac{1}{N}\sum_{i=1}^{N}(w_1 x_1^{(i)} + w_2 x_2^{(i)} + b - t^{(i)})^2$$

In [31]:

```
# Non-vectorized implementation
def cost(w1, w2, b, X, t):
    '''
    Evaluate the cost function in a non-vectorized manner for
    inputs `X` and targets `t`, at weights `w1`, `w2` and `b`.
    '''
    costs = 0
    for i in range(len(t)):
        # y_i = w1 * X[i, 0] + w2 * X[i, 0] + b   # for single feature of input data
        y_i = w1 * X[i, 0] + w2 * X[i, 1] + b     # for two features of input data
        # y_i = w1 * X[i] + w2 * X[i] + b              # All features of input data
        t_i = t[i]
        costs += (y_i - t_i) ** 2
    return costs / len(t)

```

In [32]:

```

cost(3, 5, 1, x_input, y_target)
```

Out[32]: 2475.821173270752

In [33]:

```

cost(3, 5, 0, x_input, y_target)
```

Out[33]: 2390.2197701086957

Vectorizing the cost function:

$$\mathcal{E}(y,t) = \frac{1}{N}\|\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{t}\|^2$$

In [35]:

```
def cost_vectorized(w1, w2, b, X, t):
    '''
    Evaluate the cost function in a vectorized manner for
    inputs `X` and targets `t`, at weights `w1`, `w2` and `b`.
    '''
    N = len(y_target)
    w = np.array([w1, w2])
    # print(w)
    y = np.dot(X, w) + b * np.ones(N)
    cost_vect = np.sum((y - t)**2) / (N)
    return cost_vect

```

In [36]:

```
cost_vectorized(3, 5, 1, x_input, y_target)
```

Out[36]: 2475.821173270751

In [37]:

```


cost(3, 5, 0, x_input, y_target)
```

Out[37]: 2390.2197701086957

Comparing Processing Speed of the Vectorized vs Nonvectorized code

We'll see below that the vectorized code already runs ~2x faster than the non-vectorized code! Hopefully this will convince you to always vectorized your code whenever possible

In [38]:

```
import time
t40 = time.time()
print(cost(3, 5, 1, x_input, y_target))
t41 = time.time()
time_CostNonvect = t41 - t40
print(time_CostNonvect)
```

```
2475.821173270752
0.0009961128234863281
```

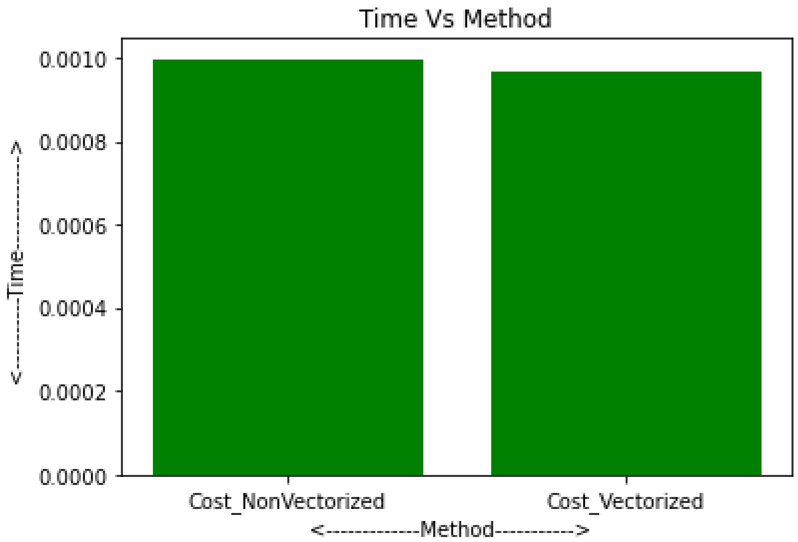
In [39]:

```
import time
t50 = time.time()
print(cost_vectorized(3, 5, 1, x_input, y_target))
t51 = time.time()
time_CostVect = t51 - t50
print(time_CostVect)
```

```
2475.821173270751
0.0009663105010986328
```

In [40]:

```python
import matplotlib.pyplot as plt
%matplotlib inline
# plt.style.use('ggplot')

x = ['Cost_NonVectorized', 'Cost_Vectorized']
Processing_Times = [time_CostNonvect, time_CostVect]


x_pos = [i for i, _ in enumerate(x)]

plt.bar(x_pos, Processing_Times, color='green')
plt.xlabel("<-------------Method----------->  ")
plt.ylabel("<----------Time-------------> ")
plt.title(" Time Vs Method ")

plt.xticks(x_pos, x)

plt.show()
```



In [ ]:

```
1
```