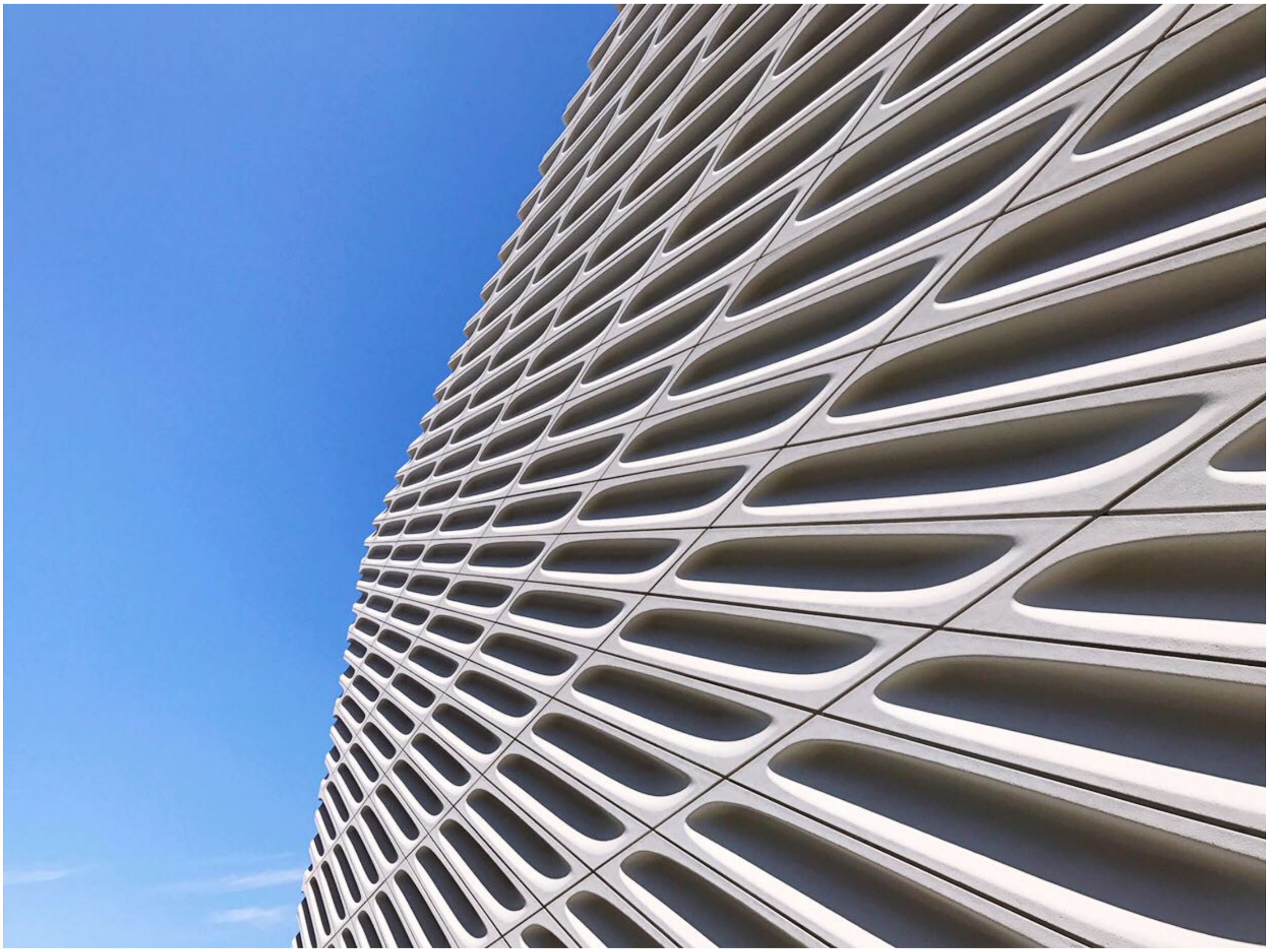
Vectorization Implementation in Machine Learning

Yang Liu

Oct 13, 2018 · 9 min read



Introduction

In machine learning filed, advanced players have the need to write their own cost function or optimization algorithm in achieving a more customized model, not just using existing machine learning libraries. In order to fully take advantage of computation power of today's computers, the state of art of implementation of algorithm is vectorizing all the computations. This allows you to achieve parallelized computation, for example fully use the processors of GPU. In this post, the implementation of vectorization of machine learing is introduced. All the code used in this post can be found in my github.

Prerequisite: Numpy Array

The most important tool we will use in this vectorization process in numpy array. Note that, we don't use numpy matrix since numpy matrix is strictly 2-D dimensional. Actually, numpy matrix is a subset of numpy array. Thus, for convenience we always use numpy array. Here is a review of numpy array arithmetic in order to better follow the later contents. Here is how to **define numpy array**:

```
# import numpy
import numpy as np

# define two numpy arrays
a = np.array([[1,2],[3,4]])
b = np.array([[1,1],[1,1]])

print(a)
>>> array([[1, 2],
           [3, 4]])
print(b)
>>> array([[1, 1],
           [1, 1]])
```

Numpy Array Addition

```
# addition
print(a + b)
```

```
>>> array([[2, 3],
           [4, 5]])
```

Numpy Array Subtraction

```
# substraction
print(a - b)

>>> array([[0, 1],
           [2, 3]])
```

Numpy Array Multiplication

Note that if directly use ‘*’ multiplication, this is different from matrix multiplication known as dot product. The ‘*’ operations between two arrays is just multiply the elements in the same position.

```
# multiplication:
print(a * b)

>>> array([[1, 2],
          [3, 4]])
```

Numpy Array Dot Product

We use numpy’s dot function to achieve matrix multiplication. A so convenient way is by just using **‘@’ symbol**, it works exactly the same way.

```
# matrix multiplication
print(np.dot(a,b))

>>> array([[1, 2],
          [3, 4]])

# matrix product alternative
print(a@b)
```

```
>>> array([[3, 3],
          [7, 7]])
```

Numpy Array Dimension

Here we show two example demonstrating how dimension works in numpy array. Note that, in the first case, it is an array with one row three columns, while in the second case, it is an array with three rows but one column.

```
# numpy array with one row
a =  np.array([1,2,3])
print(a.shape)

>>> (3,)

# numpy array with three rows
b = np.array([[1],[2],[3]])
print(b.shape)

>>> (3, 1)
```

Numpy Array Indexing and Slicing

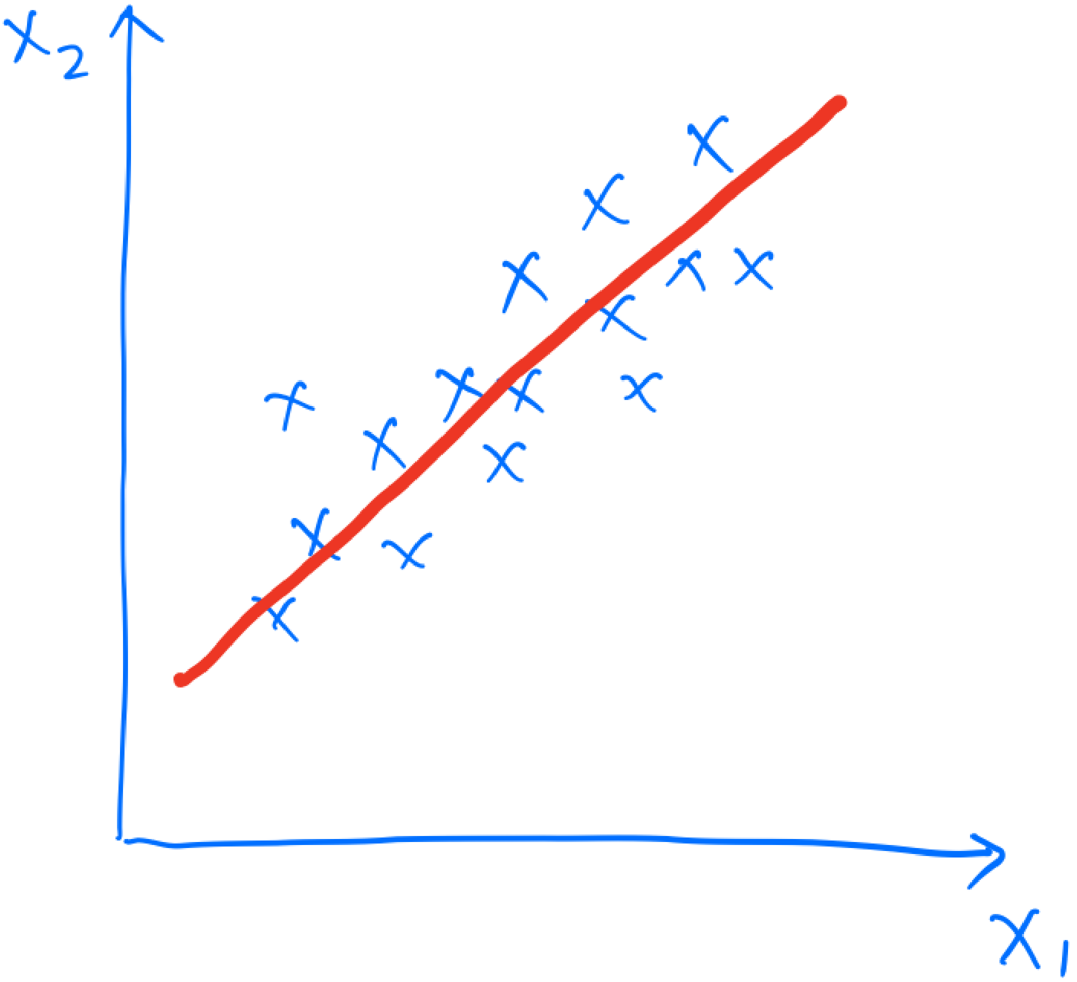
For 2D numpy array, when we slicing one element in an array denoted as A, we can use A[i, j] where i is the row index and j is the column index. If want to select a whole row i, use A[i, :], similarly for selecting a whole column j using A[:,j].

```
# Define an 3x3 2d array
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a)

>>> array([[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]])

# select first element in the array
print(a[0,0])

>>> 1
```

```
# select first row of the array
print(a[0,:])

>>> array([1, 2, 3])

# select second coulumn of the array
print(a[:,1])

>>> array([2, 5, 8])
```

Prerequisite: Linear Regression Cost Function

In this section, we will review some concepts and its mathematical expressions of linear regression. Since we need to use these formulas to achieve gradient descent algorithm in the next section to see how to implement vectorization.



The hypothesis of linear regression is defined as:

$$h_\theta(x^{(i)}) = \theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \ldots + \theta_j x_j^{(i)}$$

The cost function of linear regression is defined as:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

The derivative of cost function to each θ is defined as:

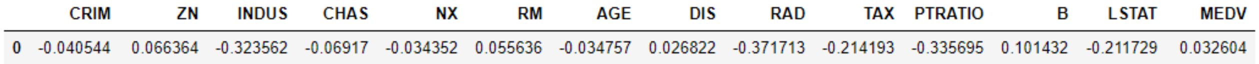
$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

In each iteration of gradient descent, we update all the θ using the following equation:

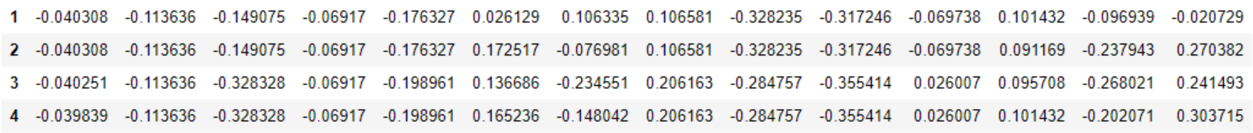
$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

DataSet

The dataset we used is 'Boston Housing' from UCI Machine Leraning Repository. It used the features like house area size, built year etc. to predict the house price in Boston area. Here is what the data looks like:



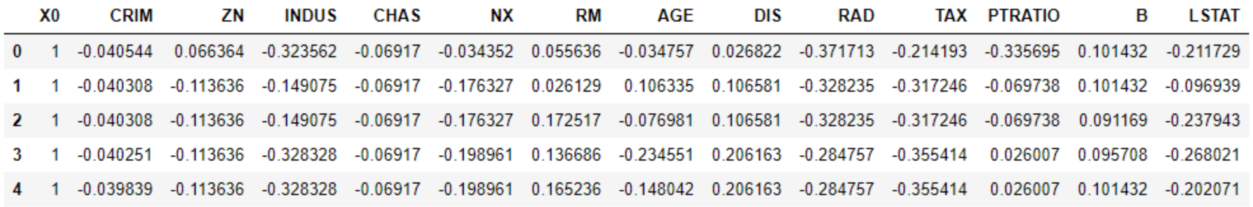
	CRIM	ZN	INDUS	CHAS	NX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	-0.040544	0.066364	-0.323562	-0.06917	-0.034352	0.055636	-0.034757	0.026822	-0.371713	-0.214193	-0.335695	0.101432	-0.211729	0.032604



1	-0.040308	-0.113636	-0.149075	-0.06917	-0.176327	0.026129	0.106335	0.106581	-0.328235	-0.317246	-0.069738	0.101432	-0.096939	-0.020729
2	-0.040308	-0.113636	-0.149075	-0.06917	-0.176327	0.172517	-0.076981	0.106581	-0.328235	-0.317246	-0.069738	0.091169	-0.237943	0.270382
3	-0.040251	-0.113636	-0.328328	-0.06917	-0.198961	0.136686	-0.234551	0.206163	-0.284757	-0.355414	0.026007	0.095708	-0.268021	0.241493
4	-0.039839	-0.113636	-0.328328	-0.06917	-0.198961	0.165236	-0.148042	0.206163	-0.284757	-0.355414	0.026007	0.101432	-0.202071	0.303715

Our dataset has 506 entries, we denote it as the number of entries as m. The number of features is n=14 including the interception feature which we initialize as all ones. See below:

```
# Insert X0 Column
Xd = df.drop(columns=['MEDV'])
Xd.insert(0, 'X0', 1)
Xd.head()
```



	X0	CRIM	ZN	INDUS	CHAS	NX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	1	-0.040544	0.066364	-0.323562	-0.06917	-0.034352	0.055636	-0.034757	0.026822	-0.371713	-0.214193	-0.335695	0.101432	-0.211729
1	1	-0.040308	-0.113636	-0.149075	-0.06917	-0.176327	0.026129	0.106335	0.106581	-0.328235	-0.317246	-0.069738	0.101432	-0.096939
2	1	-0.040308	-0.113636	-0.149075	-0.06917	-0.176327	0.172517	-0.076981	0.106581	-0.328235	-0.317246	-0.069738	0.091169	-0.237943
3	1	-0.040251	-0.113636	-0.328328	-0.06917	-0.198961	0.136686	-0.234551	0.206163	-0.284757	-0.355414	0.026007	0.095708	-0.268021
4	1	-0.039839	-0.113636	-0.328328	-0.06917	-0.198961	0.165236	-0.148042	0.206163	-0.284757	-0.355414	0.026007	0.101432	-0.202071

```
# numpy array format
X = Xd.values
y = df.MEDV.values

# sample size
m = len(df.index)
print(m)

>>> 506

# number of features
n = X.shape[1]
print(n)

>>> 14
```

For Loop V.S. Vectorization

In this section, we do step by step comparison of for loop and vectorization method by applying gradient descent algorithm for linear regression. We compare the running time for each implementation of the formulas in the linear regression section.

We initialize all the thetas as ones:

```
# Initialize theta
theta = np.ones(n)

print(theta)
>>> array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.])
```

Hypothesis Implementation: For Loop

In order to achieve the hypothesis function of linear regression, if we use for loop, it can be achieved using the following code:

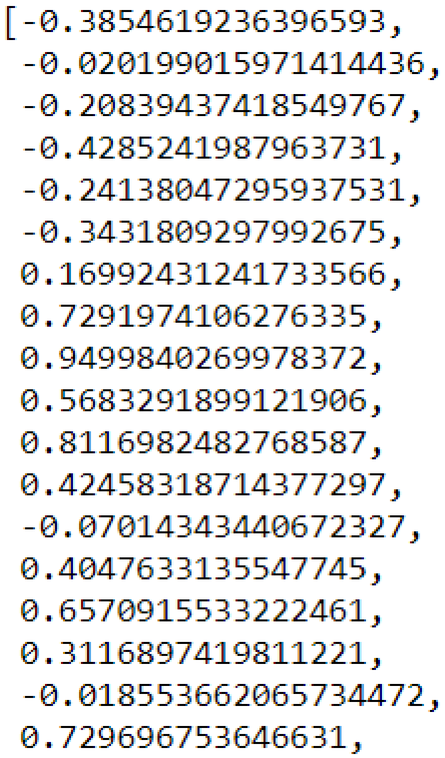
```
# hypothesis for the first sample
hypo = 0
for j in range(n):
    hypo += theta[j]*X[0,j]
```

In order to get hypothesis for each of the sample, we need a list to store it and another for loop to iterate over all the samples:

```
%%time
# hypothesis for all the samples
all_hypo = []
for i in range(m):
    hypo_i = 0
    for j in range(n):
        hypo_i += theta[j]*X[i,j]
    all_hypo.append(hypo_i)

>>> Wall time: 4 ms
```

We can see the running time is 4 ms, it is not too crazy since this implementation is simple enough and the dataset is small.

The result is show as:



```
[-0.3854619236396593,
 -0.020199015971414436,
 -0.20839437418549767,
 -0.4285241987963731,
 -0.24138047295937531,
 -0.3431809297992675,
 0.16992431241733566,
 0.7291974106276335,
 0.9499840269978372,
 0.5683291899121906,
 0.8116982482768587,
 0.42458318714377297,
 -0.07014343440672327,
 0.4047633135547745,
 0.6570915533222461,
 0.3116897419811221,
 -0.018553662065734472,
 0.729696753646631,
```

Hypothesis Implementation: Vectorization

The hypothesis of each sample can be vectorized using following formula:

$$h_\theta(x^{(i)}) = \theta^T x^{(i)} = \begin{bmatrix} \theta_0 & \theta_1 & \ldots & \theta_n \end{bmatrix} \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \ldots \\ x_n^{(i)} \end{bmatrix}$$

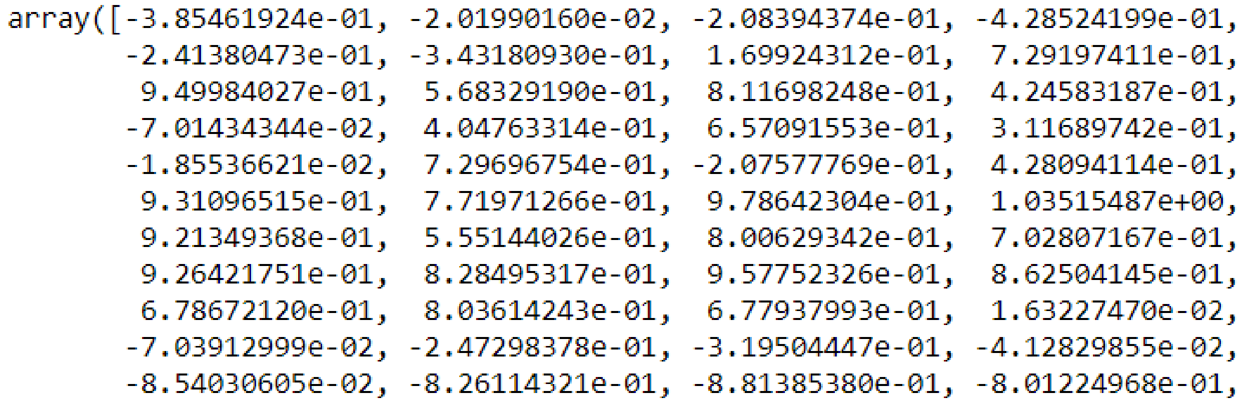
In order to achieve the hypothesis for all the samples as a list, we use the following array dot product:

$$h_\theta(x) = X\theta = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & \ldots & x_n^{(1)} \\ x_0^{(2)} & x_1^{(2)} & & x_n^{(2)} \\ \vdots & \vdots & \ldots & \vdots \\ x_0^{(m)} & x_1^{(m)} & \ldots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

The code achievement is pretty easy and clean:

```
%%time
# matrix format
hypo = X@theta

>>> Wall time: 0 ns
```

We can find that it needs tiny amount of time to calculate and even can not be shown. The calculation results is shown as below. Compared with the for loop results, we can see we obtain exactly the same results.



```
array([-3.85461924e-01, -2.01990160e-02, -2.08394374e-01, -4.28524199e-01,
       -2.41380473e-01, -3.43180930e-01,  1.69924312e-01,  7.29197411e-01,
        9.49984027e-01,  5.68329190e-01,  8.11698248e-01,  4.24583187e-01,
       -7.01434344e-02,  4.04763314e-01,  6.57091553e-01,  3.11689742e-01,
       -1.85536621e-02,  7.29696754e-01, -2.07577769e-01,  4.28094114e-01,
        9.31096515e-01,  7.71971266e-01,  9.78642304e-01,  1.03515487e+00,
        9.21349368e-01,  5.55144026e-01,  8.00629342e-01,  7.02807167e-01,
        9.26421751e-01,  8.28495317e-01,  9.57752326e-01,  8.62504145e-01,
        6.78672120e-01,  8.03614243e-01,  6.77937993e-01,  1.63227470e-02,
       -7.03912999e-02, -2.47298378e-01, -3.19504447e-01, -4.12829855e-02,
       -8.54030605e-02, -8.26114321e-01, -8.81385380e-01, -8.01224968e-01,
```

Cost function Implementation: For Loop

Based on the results we obtained from the hypothesis, we need another loop to iterate over all the samples to calculate cost function.

```
%%time
# cost function
cost = 0
for i in range(m):
    hypo_i = 0
    for j in range(n):
        hypo_i += theta[j]*X[i,j]
    cost_i = (hypo_i - y[i])**2
    cost += cost_i
cost = (1/(2*m))*cost

>>> Wall time: 4 ms
```

The running time is 4 ms and the result is showing as:

```
print(cost)

>>> 1.399752908228425
```

Cost function Implementation: Vectorization

Based on the vectorization of hypothesis, we can easily vectorize the cost function as:

$$J(\theta) = \frac{1}{2m}(X\theta - y)^T(X\theta - y)$$

The code implementation is still very clean:

```
%%time
# cost function
cost = (1/(2*m))*np.transpose((X@theta - y))@(X@theta - y)

>>> Wall time: 0 ns
```

Again, this vectorized calculation is instantly. The calculated result is identical to the for loop result:

```
print(cost)

>>> 1.3997529082284244
```

Derivation Implementation: For Loop

Calculate the derivation of cost function to a specified θ is coded as following:

```
dev_sum = 0
for i in range(m):
    hypo_i = 0
    for j in range(n):
        hypo_i += theta[j]*X[i,j]
    dev_i = (hypo_i - y[i])*X[i,k]
    dev_sum += dev_i
dev_sum = (1/m)*dev_sum
```

To calculate the derivation to all θ and output a list, we need another for loop iterate over all the columns:

```
%%time
# derivation
dev_list = []
for k in range(n):
    dev_sum = 0
    for i in range(m):
        hypo_i = 0
        for j in range(n):
            hypo_i += theta[j]*X[i,j]
        dev_i = (hypo_i - y[i])*X[i,k]
        dev_sum += dev_i
    dev_sum = (1/m)*dev_sum

    dev_list.append(dev_sum)

>>> Wall time: 47 ms
```

The running time is 47 ms, as we have more loops, the time cost difference of for loop and vectorization starts to become significant.

The for loop derivation results is:

```
print(dev_list)

>>> [0.9999999999999983,
 0.07814620360307895,
 -0.11042922261438312,
 0.2620302340552936,
 0.05504439083525137,
 0.23892542562534522,
 -0.06454255823702795,
 0.2611634394125097,
 -0.1453677181065729,
 0.43106386997897883,
 0.38303455280215737,
 0.16591512402899725,
 -0.09920797306076046,
 0.1835280968258358]
```

Derivation Implementation: Vectorization

The derivation of cost function regards to each θ can be vectorized as:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}(x_j)^T(X\theta - y)$$

The derivation of cost function to all θ can be vectorized as:

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m}X^T(X\theta - y)$$

The code implementation is still super clean:

```
%%time
dev = (1/m)*np.transpose(X)@(X@theta - y)

>>> Wall time: 999 µs
```

The direct comparison is 999 µs v.s. 47 ms. Here is the vectorization calculation results:

```
print(dev)

array([ 1.        ,  0.0781462 , -0.11042922,  0.26203023,
0.05504439, 0.23892543, -0.06454256,  0.26116344, -0.14536772,
0.43106387, 0.38303455,  0.16591512, -0.09920797,  0.1835281 ])
```

Again the results is the same for the two methods.

Put everything together: Optimization

In the section, we use all the implementations we developed before and write a gradient descent iteration to compare the two methods.

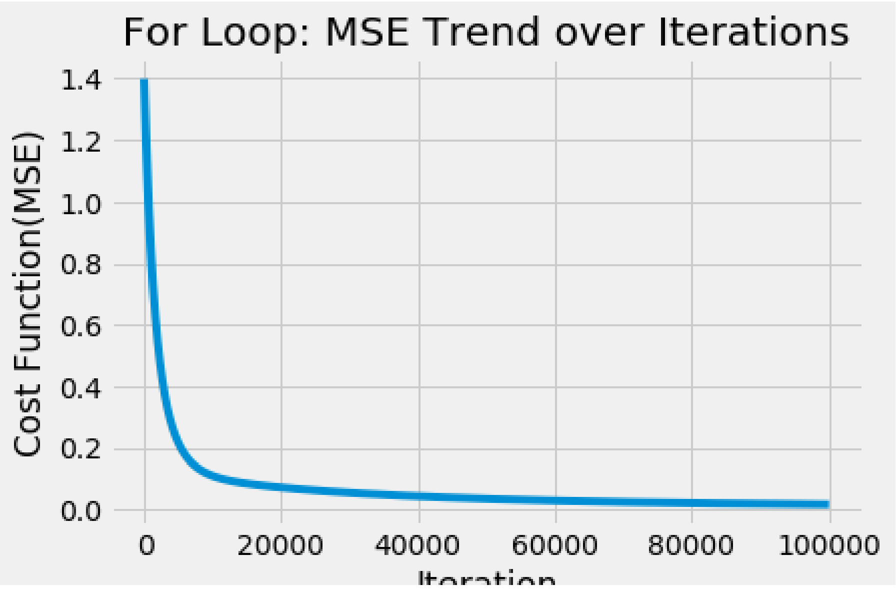
Gradient Descent: For Loop

In order to achieve descent optimization results, we set the iteration times to be 100 thousands. We need nested four for loops in order to achieve the gradient descent algorithm. The learning rate is set to be 0.0005 and the thetas are initialized as all ones. The code is shown as below:

```
%%time
a = 0.0005
theta = np.ones(n)

cost_list = []

for itr in range(100000):

    dev_list = []
    for k in range(n):
        dev_sum = 0
        for i in range(m):
            hypo_i = 0
            for j in range(n):
                hypo_i += theta[j]*X[i,j]
            dev_i = (hypo_i - y[i])*X[i,k]
            dev_sum += dev_i
        dev_sum = (1/m)*dev_sum

dev_list.append(dev_sum)

    theta = theta - a*np.array(dev_list)

    cost_val = cost_loop(theta)

    cost_list.append(cost_val)

>>> Wall time: 1h 15min 58s
```

The total running time is 1 hour and 15 mins. The following is the minimum cost we obtained. We also provide a plot showing the cost function changes with respect to iterations.

```
print(cost_val)

>>> 0.017663350184258856
```



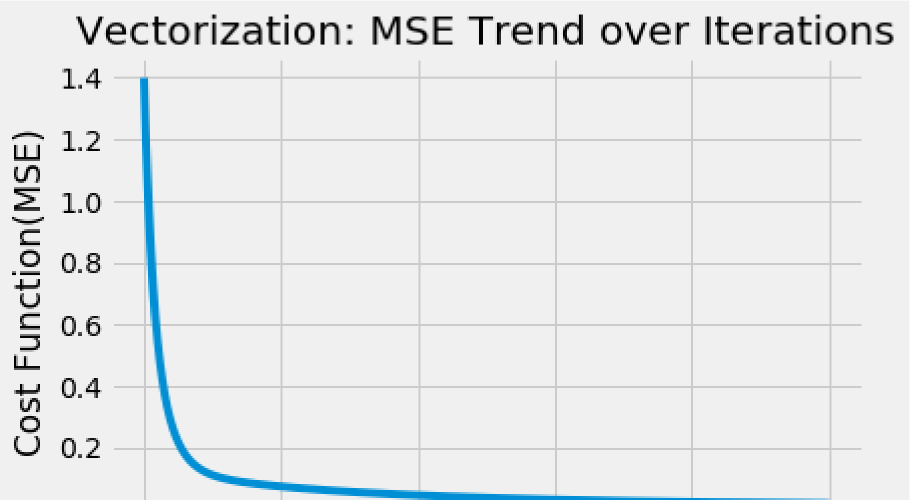
Gradient Descent: Vectorization

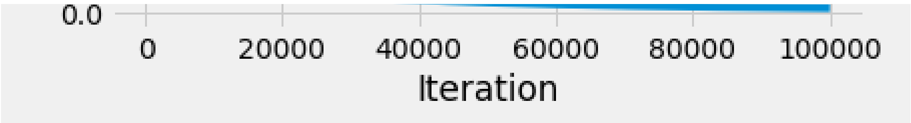
The implementation of vectorized gradient descent is super clean and elegant.

```
%%time
a = 0.0005
theta = np.ones(n)

cost_list = []

for i in range(100000):

    theta = theta - a*(1/m)*np.transpose(X)@(X@theta - y)

    cost_val = cost(theta)
    cost_list.append(cost_val)

>>> Wall time: 1.75 s
```

The vectorized approach has the minimum cost function value as below. Again, the cost changes with respect to iterations is provided:

```
print(cost_val)

>>> 0.017663350184258835
```

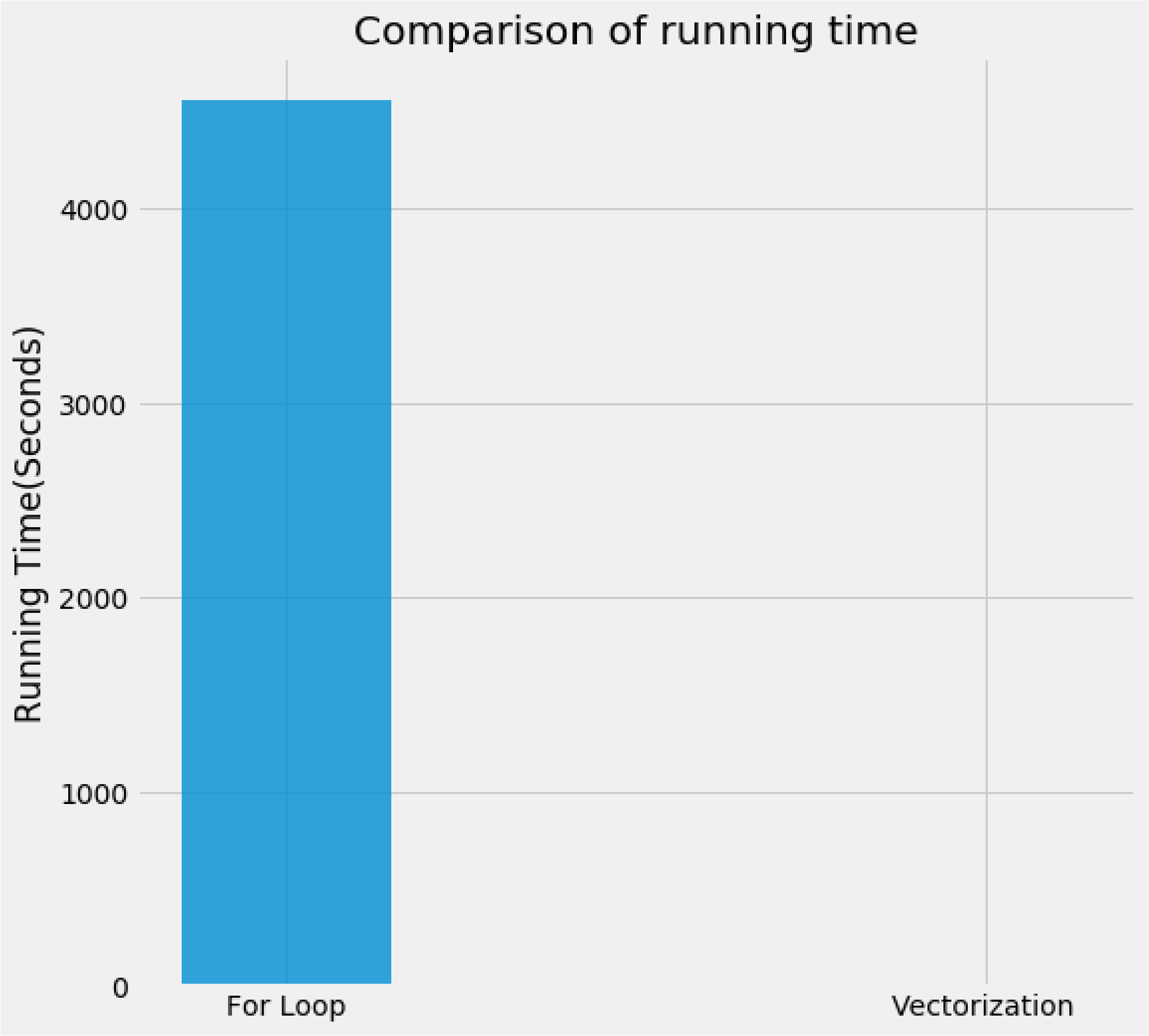




We can see the minimum cost value of the two approaches are almost exactly the same. But the algorithm implementation time is 1.75 seconds using vectorization versus 1 h and 15 mins using for loop.

Conclusion

Here is a plot showing the running time difference of the two approaches implementing the same algorithm and using exactly the same learning rate and initial θ values. The two approaches achieved the same accuracy. However, the vectorization approach cost 1.75 seconds while the for loop cost 4558 seconds. The vectorization approach is 2600 times faster than the for loop approach.



The time complexity of vectorization method is O(s), where s in the number of iterations. As contrast, the for loop approach time complexity is O(s*n*m*n), where s is the iterations, m is the dataset sample number, n is the dataset feature number. In this case, our dataset is small enough with m=506 and n=14, however we observed such a huge difference in time complexity. Imaging for big data, how huge this difference would be. As noway's computers and GPUs are made of thousands 'cores', and we can even have multiple GPUs or use a cluster of computers, we need to take good advantage of these computation power. And the way is implementing your algorithm by parallel computation. Therefore, using vectorization in our machine leaning algorithm is the key to boost your algorithm and save you a huge amount of training time. Vectorization would be a great approach we need to consider and worth to spent time on.

Machine Learning Vectorization Parallel Computing Linear Regression Gradient Descent