Gradient Descent Vector/Matrix Implementation

Neba Nfonsang

University of Denver

Here is the goal, we want to implement the batch gradient descent rule in a vectorized/matrix manner. The gradient descent rule is:

Gradient Descent

Repeat until converge {

$$\theta_j = \theta_j + \alpha * \frac{1}{n} * \sum_{i=1}^{n} [(y^{(i)} - \mathbf{h}(x^{(i)})) * x_j^{(i)}]$$

} for every parameter j=0,1…k

Highlighted below are important packages you would find useful in carrying out matrix or vector operations for your Gradient Descent implementations

```
import pandas as pd
import numpy as np
from patsy import dmatrix
from numpy.linalg import inv

from sklearn.linear_model import SGDRegressor

import statsmodels.api as sm

import matplotlib.pyplot as plt
```

Here is your data

```
data = pd.read_csv("Health_Data.csv")
data
```

	age	income	illness	reduced	health
0	0.19	0.55	1	4	1
1	0.19	0.45	1	2	1
2	0.19	0.90	3	0	0

Use the dmatrix function to create a design matrix X for the input data

```
X = dmatrix("age + income + illness + reduced", data)[:]
X
```

```
array([[1.  , 0.19, 0.55, 1.  , 4.  ],
       [1.  , 0.19, 0.45, 1.  , 2.  ],
       [1.  , 0.19, 0.9 , 3.  , 0.  ],
       ...,
       [1.  , 0.37, 0.25, 1.  , 0.  ],
       [1.  , 0.52, 0.65, 0.  , 0.  ],
       [1.  , 0.72, 0.25, 0.  , 0.  ]])
```

This is how matrix multiplication works in Python:


```
A = np.array([1, 2])
A
```

```
array([1, 2])
```

```
B = np.array([[1, 1],
              [2, 2]])
B
```

```
array([[1, 1],
       [2, 2]])
```

```
np.matmul(A, B)
```

```
array([5, 5])
```

Now to implement Batch gradient descent in a vectorized/matrix manner, we need to compute the errors, which is the bottom line of the gradient descent algorithm, but to compute the errors we first need to compute the predicted values since **error = pred_y – y.**

How to compute predicted y using matrix multiplication

Here are predicted y values for each instance, from instance 1 to instance n.

$$
\begin{aligned}
y_1 &= \beta_0 + \beta_1 x_1 + \epsilon_1 \\
y_2 &= \beta_0 + \beta_1 x_2 + \epsilon_2 \\
&\vdots \\
y_n &= \beta_0 + \beta_1 x_n + \epsilon_n
\end{aligned}
$$

Here is a vectorized/matrix computation of the predicted values for all the instances

$$
\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}
$$

$$
Y = X\beta + \varepsilon
$$

That means, predicted y is $\hat{\boldsymbol{y}} = \boldsymbol{X\beta}$**,** which is a matrix multiplication of the X design matrix and the **column** vector of beta values.

Note that $\hat{y} = X\beta$, **is the same as** $\hat{y} = (\beta_{row})(X.T)$ where X.T is the transpose of the design matrix X and β_{row} *is a row vector* of beta values. So you can use one of these formulas or the other.

- If you are using $\hat{y} = X\beta$, make sure that β *is a column vector by reshaping it using* $.reshape(k, 1)$; where k is the number beta values in the β ***vector*** (b0 is also counted)
- If you are using $\hat{y} = (\beta_{row})(X.T)$ make sure the beta values are reshaped into a row vector using .reshape(1, k)

For example, if your betta values were 1, 2, 3 ,4 5, then:

```
# predicted y through matrix multiplication example
b= np.array([1, 3, 2, 4, 5]).reshape(1, 5)
np.matmul(b, X.T)
```

How to compute the errors for each instance in the data

```
]: # compute errors example
   y_pred = np.matmul(b, X.T)
   errors = y - y_pred
   errors

]: array([[-25.67, -15.47, -15.37, ...,  -5.61,  -3.86,  -3.66]])
```

Note that the output for the errors here is a row vector

Here is the gradient descent rule:

Gradient Descent

Repeat until converge {

$$\theta_j = \theta_j + \alpha * \frac{1}{n} * \sum_{i=1}^{n} [(y^{(i)} - h(x^{(i)})) * x_j^{(i)}]$$

} for every parameter j=0,1…k

1. According the gradient rule, for each variable in the design matrix, we multiply the errors with the value of the variable (error*x).
2. Then for each variable, sum error*x over all the instances
3. Then multiply the result by 1/n then by alpha.
4. Whatever result you get in 3, add that to parameter values to get the new parameter values:

   All these steps can be implemented into a single line of code

This is how to multiply the errors to each value of x in the X matrix

```
# compute error*X
errors.T*X
```

```
array([[ -25.67  ,   -4.8773,  -14.1185,  -25.67  , -102.68  ],
       [ -15.47  ,   -2.9393,   -6.9615,  -15.47  ,  -30.94  ],
       [ -15.37  ,   -2.9203,  -13.833 ,  -46.11  ,   -0.    ],
       ...,
       [  -5.61  ,   -2.0757,   -1.4025,   -5.61  ,   -0.    ],
       [  -3.86  ,   -2.0072,   -2.509 ,   -0.    ,   -0.    ],
       [  -3.66  ,   -2.6352,   -0.915 ,   -0.    ,   -0.    ]])
```

We are transposing the errors before multiplying to X because the errors we got were row vectors and we want to do an element wise multiplication of the errors to the values of each variable.

You can then use np.sum(errors.T*X), but make sure you are setting this to the right axis to sum over all instances or individuals not summing over all variables.

Now that you know how this works, you will implement this inside a loop until convergence (remember you are implementing a batch gradient descent in a vectorized/matrix way):

Gradient Descent

Repeat until converge {

$$\theta_j = \theta_j + \alpha * \frac{1}{n} * \sum_{i=1}^{n} [(y^{(i)} - \mathbf{h}(x^{(i)})) * x_j^{(i)}]$$

} for every parameter j=0,1…k

Blessings!