Gradient Descent and Simple Linear Regression
Logic and Python tutorial

Sophie Marchand

May 2020

1 Shortcut

Gradient descent is an optimization algorithm employed to find the parameters $\mathbf{v}$ of a function f minimizing a cost function C. Its iterative procedure is as follow:

1. Initialize parameters to random small values
2. Calculate the cost function C over the all training data
3. Compute the update of the parameters $\mathbf{v}$ with $\mathbf{v} - \eta \nabla C(\mathbf{v})$ with η the learning rate
4. Repeat the steps 2 and 3 until reaching "good" enough parameters

This procedure is for one pass of the batch gradient descent. For the stochastic one, step 1 includes a randomization of the training data and step 2 is performed over one instance selected according to the algorithm iteration. Then, the steps 2 and 3 are performed for each randomized training input.

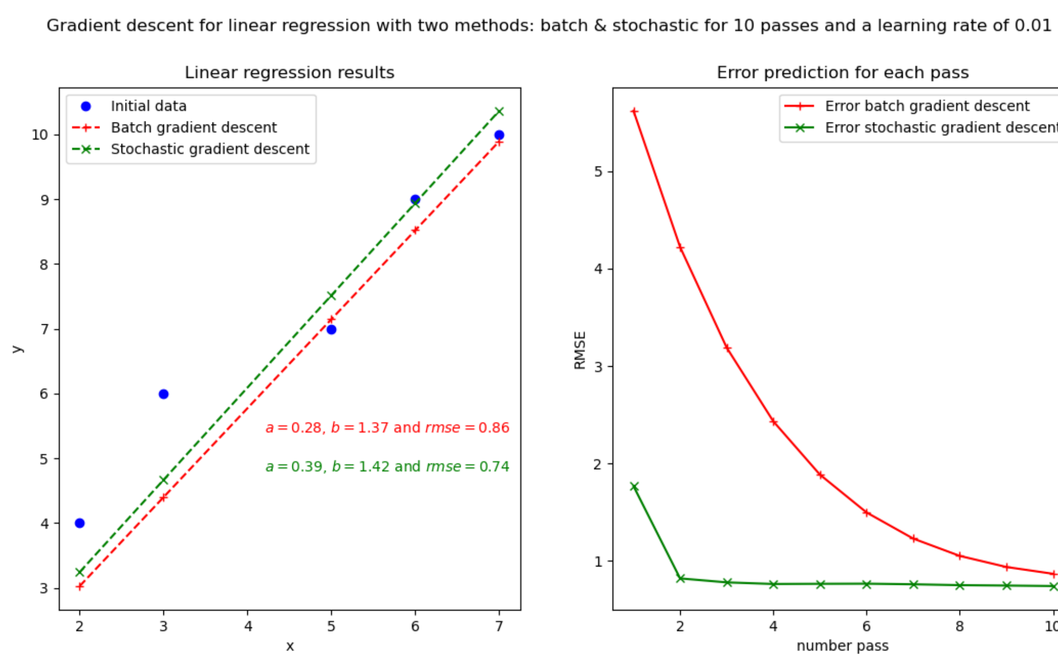

Figure 1: Output of the tutorial in section 3

2 Logic details

We show here the logic behind the gradient descent applied to a simple linear regression problem. The objective of this regression is to find the optimal slop b and intercept a which verify $y = a + b \times x$ while minimizing the prediction error for a set of n points (x, y). In this work, the error function chosen is the Sum of Squared Residuals (SSR) defined by equation (1) where $\mathbf{v}$ is the vector of coefficients $\begin{pmatrix} a \\ b \end{pmatrix}$ and y_{perd} the predicted output variable.

$$SSR(\mathbf{v}) = \frac{1}{2n} \sum_{i=1}^{n} (y_{perd}(\mathbf{v}, i) - y_i)^2 \tag{1}$$

Using the Taylor series expansion on $SSR(\mathbf{v})$, we obtain the equation (2). Then, replacing ϵ by $-\eta \nabla SSR(\mathbf{v})$ with η a small positive value called learning rate, we have the expression (3).

$$SSR(\mathbf{v} + \epsilon) \approx SRS(\mathbf{v}) + \epsilon^T \nabla SSR(\mathbf{v}) \tag{2}$$

$$SSR(\mathbf{v} - \eta \nabla SSR(\mathbf{v})) \approx SSR(\mathbf{v}) - \eta \nabla SSR(\mathbf{v})^2 \leq SSR(\mathbf{v}) \tag{3}$$

We deduce from the previous expression that updating $\mathbf{v}$ by $\mathbf{v} - \eta \nabla SSR(\mathbf{v})$ may reduce the value of $SSR(\mathbf{v})$. This is the logic adopted by the gradient descent method consisting in the following steps:

1. Initiate the values of $\mathbf{v}$ to zero or small random values

   [Stochastic gradient descent] Randomized the training data order giving the order array r

2. Compute the prediction error $(y_{perd}(\mathbf{v}, i) - y_i)$ for:

   [Batch gradient descent] all training data $\mathbf{i}$ before calculating the update

   [Stochastic gradient descent] each training data instance $\mathbf{i}$ and calculate the update immediately

3. Compute the update of $\mathbf{v}$ with:

   [Batch gradient descent] $i \subset [1, n]$

$$\begin{cases} a = a - \eta \frac{\partial SSR(\mathbf{v},i)}{\partial a} = a - \frac{\eta}{n} \sum_{i=1}^{n} (y_{perd}(\mathbf{v}, i) - y_i) \\ b = b - \eta \frac{\partial SSR(\mathbf{v},i)}{\partial b} = b - \frac{\eta}{n} \sum_{i=1}^{n} (y_{perd}(\mathbf{v}, i) - y_i) x_i \end{cases} \tag{4}$$

   [Stochastic gradient descent] $i = r[j]$ with j the iteration of the gradient descent

$$\begin{cases} a = a - \eta \frac{\partial SSR(\mathbf{v},r[j])}{\partial a} = a - \eta (y_{perd}(\mathbf{v}, r[j]) - y_{r[j]}) \\ b = b - \eta \frac{\partial SSR(\mathbf{v},r[j])}{\partial b} = b - \eta (y_{perd}(\mathbf{v}, r[j]) - y_{r[j]}) x_{r[j]} \end{cases} \tag{5}$$

4. Repeat the steps 2 and 3 until reaching "good" enough coefficients. The performance threshold th_p could be defined as value on the Root Mean Square Error ($RMSE$) such that we should verify:

$$RMSE = \sqrt{\frac{\sum_{i=0}^{n} (y_i^{pred} - y_i)^2}{n}} < th_p \tag{6}$$

Remarks: the stochastic gradient descent is preferred to the batch one for large datasets. To note also that stochastic gradient descent will require a small number of passes through the dataset to reach "good" enough coefficients typically between 1-to-10 passes.

3 Python tutorial

The code source displayed below can be found on GitHub under Python_GradientDescentLinearRegression.py

```python
"""
Tutorial Gradient Descent for Linear Regression

Author: Sophie Marchand
"""
import matplotlib.pyplot as plt
import numpy as np


# Function initialization parameters, update parameters and compute error
def init_parameters():
    return 0, 0


def randomized_training_data(x, y):
    index_array = np.arange(len(x))
    np.random.shuffle(index_array)
    return x[index_array], y[index_array]


def update_parameters_batch(a, b, x, y, learning_rate):
    a_update = a - learning_rate * sum((a + b * x) - y) / len(x)
    b_update = b - learning_rate * sum(((a + b * x) - y) * x) / len(x)
    return a_update, b_update


def update_parameters_stochastic(a, b, x, y, learning_rate, iteration):
    x_it = x[iteration]
    y_it = y[iteration]
    a_update = a - learning_rate * ((a + b * x_it) - y_it)
    b_update = b - learning_rate * ((a + b * x_it) - y_it) * x_it
    return a_update, b_update


def compute_error_rmse(a, b, x, y):
    return np.sqrt(sum(np.square((a + b * x) - y)) / len(x))


# Create a set of points (x,y) and set a learning rate
x = np.array([2, 3, 5, 6, 7])
y = np.array([4, 6, 7, 9, 10])
learning_rate = 0.01
number_batch = 10
rmse_batch = []
rmse_stochastic = []

# Workflow batch gradient descent to estimate (a,b) parameters of y = a + b*x
a, b = init_parameters()
for batch in range(number_batch):
    a_update, b_update = update_parameters_batch(a, b, x, y, learning_rate)
    rmse_batch.append(compute_error_rmse(a_update, b_update, x, y))
    a, b = a_update, b_update
a_batch, b_batch = a, b

# Workflow stochastic gradient descent to estimate (a,b) parameters of y = a + b*x
a, b = init_parameters()
for batch in range(number_batch):
    x_rand, y_rand = randomized_training_data(x, y)
    for iteration in range(len(x)):
        a_update, b_update = update_parameters_stochastic(a, b, x_rand, y_rand,
    learning_rate, iteration)
        a, b = a_update, b_update
    rmse_stochastic.append(compute_error_rmse(a, b, x, y))
a_stochastic, b_stochastic = a, b
```

```

# Print results
y_batch = a_batch + b_batch * x
y_stochastic = a_stochastic + b_stochastic * x

fig, (ax1, ax2) = plt.subplots(1, 2)
fig.suptitle('Gradient descent for linear regression with two methods: batch & stochastic for '
             + str(number_batch) + ' passes and a learning rate of ' + str(learning_rate), fontsize=12)
ax1.plot(x, y, 'bo', label='Initial data')
ax1.plot(x, y_batch, linestyle='--', marker='+',
         color='r', label='Batch gradient descent')
ax1.plot(x, y_stochastic, linestyle='--', marker='x',
         color='g', label='Stochastic gradient descent')
ax1.set(xlabel='x', ylabel='y',
        title='Linear regression results')
ax1.legend()
ax1.text(4.2, 5.4, r'$a=$' + str(a_batch)[:4] +
         ', $b=$' + str(b_batch)[:4] + ' and $rmse=$' + str(rmse_batch[-1])[:4], fontsize=10, color='r')
ax1.text(4.2, 4.8, r'$a=$' + str(a_stochastic)[:4] + ', $b=$' + str(b_stochastic)[:4] +
         ' and $rmse=$' + str(rmse_stochastic[-1])[:4], fontsize=10, color='g')
ax2.plot(range(1, len(rmse_batch)+1), rmse_batch, 'r+-', label='Error batch gradient descent')
ax2.plot(range(1, len(rmse_stochastic)+1), rmse_stochastic, 'gx-', label='Error stochastic gradient descent')
ax2.set(xlabel='number pass', ylabel='RMSE',
        title='Error prediction for each pass')
ax2.legend()
plt.show()
```