Python Data Products

Course 2: Design thinking and predictive pipelines

Lecture: gradient descent in Python

Learning objectives

In this lecture we will...

- Show how gradient descent can be implemented in Python
- Introduce the relationship between equations/mathematical objectives (theory) and their implementation (practice)

Goal: Regression objective

$$\arg\min_\theta \sum_i (x_i \cdot \theta - y_i)^2$$

$$\frac{\partial f}{\partial \theta_k} = \sum_i 2X_{ik}(X_i \cdot \theta - y_i)$$

Let's look at implementing this on the same PM2.5 dataset from our previous lecture on regression

Code: Reading the data

Reading the data from CSV, and discarding missing entries:

In [1]:
```
path = "datasets/PRSA_data_2010.1.1-2014.12.31.csv"
f = open(path, 'r')
```

In [2]:
```
dataset = []
header = f.readline().strip().split(',')
for line in f:
    line = line.split(',')
    dataset.append(line)
```

In [3]:
```
header.index('pm2.5')
```

Out[3]:
```
5
```

In [4]:
```
dataset = [d for d in dataset if d[5] != 'NA']
```

Code: Extracting features from the data

Extract features from the dataset:

```
In [5]: def feature(datum):
            feat = [1, float(datum[7])] # Temperature
            return feat
```

Offset and temperature

```
In [6]: X = [feature(d) for d in dataset]
        y = [float(d[5]) for d in dataset]
```

```
In [7]: X[0]
Out[7]: [1, -4.0]
```

```
In [8]: K = len(X[0])
        K
Out[8]: 2
```

K = number of feature dimensions

Code: Initialization

Initialize parameters (and include some utility functions)

```
In [9]: theta = [0.0]*K
```

```
In [10]: theta[0] = sum(y) / len(y)
```

```
In [11]: def inner(x,y):
             return sum([a*b for (a,b) in zip(x,y)])
```

```
In [12]: def norm(x):
             return sum([a*a for a in x]) # equivalently, inner(x,x)
```

- Initializing theta_0 (the offset parameter) to the mean value will help the model to converge faster
- Generally speaking, initializing gradient descent algorithms with a "good guess" can help them to converge more quickly

Code: Derivative

Compute partial derivatives for each dimension:

```
In [13]: def derivative(X, y, theta):
             dtheta = [0.0]*len(theta)
             K = len(theta)
             N = len(X)
             MSE = 0
             for i in range(N):
                 error = inner(X[i],theta) - y[i]
                 for k in range(K):
                     dtheta[k] += 2*X[i][k]*error/N
                 MSE += error*error/N
             return dtheta, MSE
```

Derivative:

$$\frac{\partial f}{\partial \theta_k} = \sum_i 2X_{ik}(X_i \cdot \theta - y_i)$$

Also compute MSE, just for utility

Code: Derivative

Compute partial derivatives for each dimension:

In [14]:
```
learningRate = 0.003
```

In [15]:
```
while (True):
    dtheta,MSE = derivative(X, y, theta)
    m = norm(dtheta)
    print("norm(dtheta) = " + str(m) + " MSE = " + str(MSE))
    for k in range(K):
        theta[k] -= learningRate * dtheta[k]
    if m < 0.01: break
```

Update in direction of derivative

Stopping condition

```
norm(dtheta) = 0.011085715419421865 MSE = 8403.627794070962
norm(dtheta) = 0.011020632851413479 MSE = 8403.627760862651
norm(dtheta) = 0.01095593237337664 MSE = 8403.627727849314
norm(dtheta) = 0.010891611742123273 MSE = 8403.627695029725
norm(dtheta) = 0.010827668727610302 MSE = 8403.627662403022
norm(dtheta) = 0.010764101112955294 MSE = 8403.627629967714
norm(dtheta) = 0.01070090669415397 MSE = 8403.627597722905
norm(dtheta) = 0.01063808328031018 MSE = 8403.62756566733
norm(dtheta) = 0.010575628693268708 MSE = 8403.627533799903
norm(dtheta) = 0.010513540767733419 MSE = 8403.627502119632
norm(dtheta) = 0.010451817351068865 MSE = 8403.627470625426
norm(dtheta) = 0.010390456303283141 MSE = 8403.627439316158
norm(dtheta) = 0.010329455497002886 MSE = 8403.627408190638
```

Code: Derivative

Read output

```
In [16]: theta
Out[16]: [107.00031826701057, -0.6803048266097109]
```

- (Almost) identical to the result we got when using the regression library in the previous lecture

Summary

Although a crude (and fairly slow) implementation, this type of approach can be extended to handle quite general and complex objectives. However it has several difficult issues to deal with:

- How to initialize?
- How to set parameters like the learning rate and convergence criteria?
- Manually computing derivatives is time-consuming – and difficult to debug

Summary of concepts

- Briefly introduced a crude implementation of gradient descent in Python
- Later, we'll see how the same operations can be supported via libraries