Python Data Products
Course 2: Design thinking and predictive pipelines

Lecture: gradient descent in Python
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

\[
\arg \min_\theta \sum_i \left( x_i \cdot \theta - y_i \right)^2
\]

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

Let’s look at implementing this on the same PM2.5 dataset from our previous lecture on regression
Reading the data from CSV, and discarding missing entries:

```python
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']
```
Extract features from the dataset:

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

X = [feature(d) for d in dataset]
y = [float(d[5]) for d in dataset]

X[0]
```

```
[1, -4.0]
```

```
K = len(X[0])
K
```

```
2
```

K = number of feature dimensions
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
Compute partial derivatives for each dimension:

```python
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:

```python
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
```

Stopping condition
Update in direction of 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