Use K-Means for prototype selection

Shuo Xu
Department of Computer Science and Engineering
University of California San Diego
s3xu@ucsd.edu

Abstract

In this project, we introduced a coordinate descent algorithm, which updates the variable vector one dimension at a time from the beginning to the last during one update. This method is proved to be more efficient and steady in terms of convergence.

1 Idea Description

Suppose our optimization problem in to minimize $L(w)$, where $w \in R^d$ and L is differentiable.

In my coordinate descent method, the update of weight vector w is performed one dimension at a time from w_1 to w_d, and each dimension is updated using their partial derivative.

Algorithm 1 My coordinate descent algorithm

procedure COORDINATEDESCENT(epoches, learning_rate)
  $i \leftarrow 0$
  for `i < epoches` **do**
    $j \leftarrow 0$
    for `j < d` **do**
      `w_grad = L(w).w_grad() // a d-dimension vector`
      `w[j] -= learning_rate * w_grad[j]`
      $j \leftarrow j + 1$.
    $i \leftarrow i + 1$

2 Convergence

One trivial case is that L(w) is convex and differentiable, the method would converge to the optimal loss. It's easy to find a counter case on those indifferentiable points where the weights cannot be updated.

3 Experimental Result

3.1 Data preprocess

The experiment is carried out on the first 2 classes of UCI wine dataset with 59 and 71 points each class.

Before training, all the data are normalized to $[0, 1]$ using following formula, where data is stored row wise.

$$data = \frac{data - min(train\_data\_set, axis = 0)}{max(train\_data\_set, axis = 0) - min(train\_data\_set, axis = 0)}$$

The performance of baseline and k-means algorithm are compared in this section.

3.2 Performance

3.2.1 Standard Logistic Regression

We performed standard logistic regression over the dataset using sklearn, where $C = 10^{20}$ to reduce the effect of the penalty term. The models finally converged to

$$L^* = 9.992007221626415 \times 10^{-16}$$

Figure 1 is the training loss curve of my coordinate descent method and the random one with learning rate of 0.3. To be fair, the two methods start from the same initial weight via sharing the same random seed. It's obvious that our proposed algorithm not only performs as good as standard gradient descent in terms of final loss when converging, but is more steady and converges faster compared to the random approach.

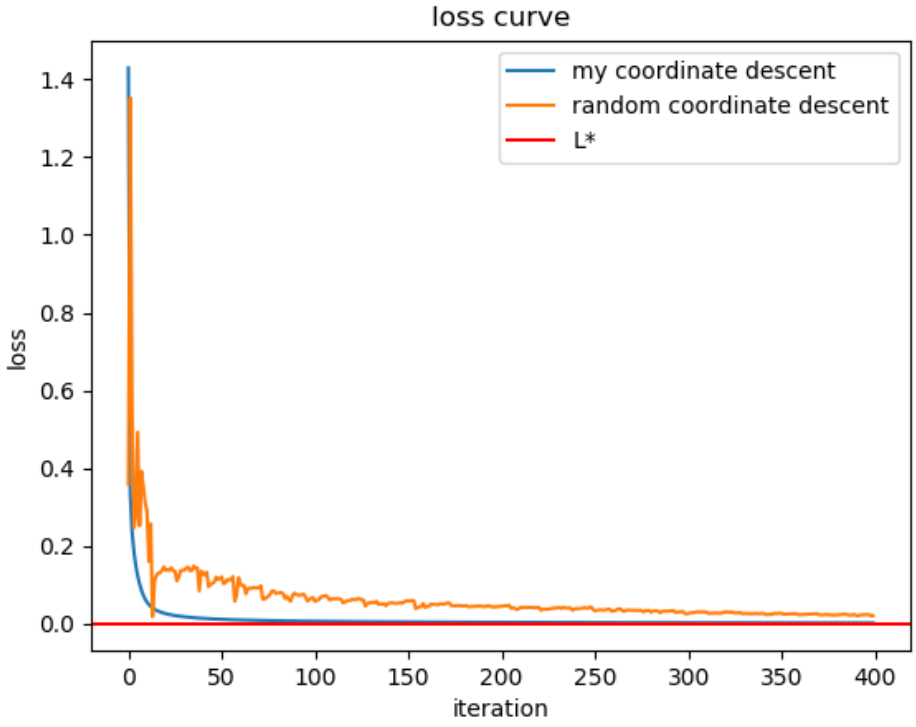

Figure 1: Training loss

4 Critical evaluation

Yes. I've got 2 ideas - greedy selection and batch selection.

Compared to the random approach, the method proposed only enforced a certain order when updating the weight. And we can further incorporate a greedy method. To be specific, after get the gradient of weight vector, we can select the dimension with the largest absolute value, which suggests that dimension would have the greatest change.

The idea of batch update is inspired by stochastic gradient descent and mini-batch gradient descent. Instead of update 1 weight at a time, we can update k dimension with the biggest absolute gradient.

Appendix of core methods implementation using Python

4.1 Base class of classifier

```
import numpy as np
from utils import onehot

class LogisticClassifier():
    def __init__(self, X, y):
        self.y_onehot = onehot(y)
        self.X, self.y = np.column_stack((np.ones(len(X)), X)), y[:,np.
    newaxis]
        self.w = None

    def logistic(self, s):
        return np.array(1 / (1 + np.exp(-s)))

    def predict(self, X):
        probs = self.logistic(np.dot(np.column_stack((np.ones(len(X)), X)
    ), self.w))
        prediction = np.zeros(probs.shape)
        prediction[probs > 0.5] = 1
        return np.squeeze(prediction)

    def prob(self, X):
        return self.logistic(np.dot(X, self.w))

    def loss(self, X, y):
        y_hat = self.logistic(np.dot(X, self.w))
        return (- np.dot(y.T, np.log(y_hat)) / len(y_hat))[0][0]

    def gradient(self, X, y):
        y_hat = self.logistic(np.dot(X, self.w))
        return np.sum((y_hat - y) * X, axis=0).reshape(-1, 1)

    def accuracy(self, test_set=None):
        if not test_set:
            test_set = self.test_set
        return np.sum(self.predict(test_set.X) == test_set.y) / len(
    test_set.y)
```

4.2 Random corrdinate Descent

```
class RandomCoordinateDescentClassifier(LogisticClassifier):
def __init__(self, X, y):
    super().__init__(X, y)

def train(self, T=600, lr=0.3):

    # initialization
    train_X, train_y = self.X, self.y
    self.train_losses = []
    np.random.seed(11)
    self.w = np.random.random((train_X.shape[1], train_y.shape[1]))
    self.W = []

    counter = 1
    num_weights = len(self.w)
    # gradient descent
    for t in range(T):
        i = np.random.randint(num_weights)
        self.w[i] -= lr * self.gradient(train_X, train_y)[i]
```

```
        self.W.append(self.w.tolist())

        # compute losses on train dataset and holdout dataset
        loss = self.loss(train_X, train_y)
        self.train_losses.append(loss)

        # save the parameters with best performance
        print("loss {} = {}".format(counter, np.array(loss)))
        counter += 1
    self.w = np.array(min(self.W, key=lambda w: self.train_losses[self.W.
    index(w)]))
    return self.train_losses
```

4.3 My corrdinate Descent

```
class MyCoordinateDescentClassifier(LogisticClassifier):
def __init__(self, X, y):
    super().__init__(X, y)

def train(self, T=600, lr=0.3):

    # initialization
    train_X, train_y = self.X, self.y
    self.train_losses = []
    np.random.seed(11)
    self.w = np.random.random((train_X.shape[1], train_y.shape[1]))
    self.W = []

    counter = 1
    # gradient descent
    for t in range(T):
        for i in range(len(self.w)):
            self.w[i] -= lr * self.gradient(train_X, train_y)[i]
        self.W.append(self.w.tolist())

        # compute losses on train dataset and holdout dataset
        loss = self.loss(train_X, train_y)
        self.train_losses.append(loss)

        # save the parameters with best performance
        print("loss {} = {}".format(counter, np.array(loss)))
        counter += 1
    self.w = self.W[np.argmin(self.train_losses)]
    return self.train_losses
```