

# Logistic Regression

**Shusen Wang**

Stevens Institute of Technology

April 12, 2019

## Abstract

Logistic regression is the most popular linear and binary classifier. This article uses logistic regression as an example to demonstrate the whole procedure of machine learning. This article covers data processing, modeling, numerical optimization, prediction, evaluation metrics, and hyper-parameter tuning. Example Python implementations are provided.

## 1 Data

Let  $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^d$  be the training features and  $y_1, \dots, y_n \in \{+1, -1\}$  be the training labels. The data are store as a feature matrix  $\mathbf{X} \in \mathbb{R}^{n \times d}$  and vector  $\mathbf{y} \in \{+1, -1\}^d$ . If the labels were in  $\{0, 1\}$ , instead of  $\{-1, 1\}$ , they must be converted to  $\{-1, 1\}$ .

For example, you can use the Diabetes dataset which has 768 samples and  $d = 8$  features. Diabetes is a binary classification dataset with labels either +1 or -1. The dataset can be downloaded from <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>.

### Feature scaling

Here we study two kinds of feature scaling: min-max normalization and standardization. Let us assume the data are unidimensional ( $d = 1$ ).

- Min-max normalization performs the transformation:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}, \quad \text{for } i = 1, \dots, n.$$

The new features  $\{x'_1, \dots, x'_n\}$  are in the interval  $[0, 1]$ .

- Standardization performs the transformation:

$$x'_i = \frac{x_i - \text{mean}(x)}{\text{std}(x)}, \quad \text{for } i = 1, \dots, n.$$

Here `mean` and `std` are the sample mean and sample standard deviation. The new features  $\{x'_1, \dots, x'_n\}$  have zero mean and unit variance.

Of course, real-world data are not unidimensional. For  $d > 1$ , we can do feature scaling for every of the  $d$  features independently. For the feature matrix  $\mathbf{X} \in \mathbb{R}^{n \times d}$ , the min-max normalization and standardization can be implemented using the following Python code.

```

1 # Min-Max Normalization
2 import numpy
3
4 d = x.shape[1]
5 xmin = numpy.min(x, axis=0).reshape(1, d)
6 xmax = numpy.max(x, axis=0).reshape(1, d)
7 xnew = (x - xmin) / (xmax - xmin)

```

```

1 # Standardization
2 import numpy
3
4 d = x.shape[1]
5 mu = numpy.mean(x, axis=0).reshape(1, d)
6 sig = numpy.std(x, axis=0).reshape(1, d)
7 xnew = (x - mu) / sig

```

Keep in mind that `xmin`, `xmax`, `mu`, and `sig` are all evaluated on the training set. We must apply the same transformation to both training and test features. The following is what we actually do in practice.

```

1 # Standardization
2 import numpy
3
4 # calculate mu and sig using the training set
5 d = x_train.shape[1]
6 mu = numpy.mean(x_train, axis=0).reshape(1, d)
7 sig = numpy.std(x_train, axis=0).reshape(1, d)
8
9 # transform the training features
10 x_train = (x_train - mu) / sig
11
12 # transform the test features
13 x_test = (x_test - mu) / sig

```

## 2 Model

Let  $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^d$  be the training features and  $y_1, \dots, y_n \in \{-1, 1\}$  be the training labels. If the labels are in  $\{0, 1\}$ , instead of  $\{-1, 1\}$ , they must be converted to  $\{-1, 1\}$ . The  $\ell_2$ -norm regularized logistic regression model is

$$\min_{\mathbf{w} \in \mathbb{R}^d} \left\{ Q(\mathbf{w}; \mathbf{X}, \mathbf{y}) \triangleq \frac{1}{n} \sum_{i=1}^n \log \left( 1 + \exp \left( -y_i \mathbf{w}^T \mathbf{x}_i \right) \right) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2 \right\}. \quad (2.1)$$

In (2.1), the average of the  $n$  terms is the loss function, the term  $\lambda \|\mathbf{w}\|_2^2$  is the regularization, and  $\lambda \geq 0$  is a tuning hyper-parameter. The function  $f(z) = \log(1 + e^{-z})$  is called logistic function, which is why the model is called logistic regression.

The following Python code computes the objective function  $Q(\mathbf{w}; \mathbf{X}, \mathbf{y})$  for given  $\mathbf{w}$ ,  $\mathbf{X}$ ,  $\mathbf{y}$ , and parameter  $\lambda$ .

```

1 # Calculate the objective function value
2 # Inputs:
3 #     w: d-by-1 matrix
4 #     x: n-by-d matrix
5 #     y: n-by-1 matrix
6 #     lam: scalar, the regularization parameter
7 # Return:
8 #     objective function value (scalar)
9 def objective(w, x, y, lam):
10     n, d = x.shape
11     yx = numpy.multiply(y, x) # n-by-d matrix
12     yxw = numpy.dot(yx, w) # n-by-1 matrix
13     vec1 = numpy.exp(-yxw) # n-by-1 matrix
14     vec2 = numpy.log(1 + vec1) # n-by-1 matrix
15     loss = numpy.mean(vec2) # scalar
16     reg = lam / 2 * numpy.sum(w * w) # scalar
17     return loss + reg

```

## 2.1 Model derivation

Logistic regression has two forms; (2.1) is one of the two. Here we derive both models using sigmoid function and cross-entropy function. The reader can skip this subsection without affecting reading the rest of this article.

First, let us recall the definitions of sigmoid function and cross-entropy function. The sigmoid function maps a real number between  $-\infty$  and  $+\infty$  to a number between 0 and 1. It is defined by

$$\text{sigmoid}(z) \triangleq \frac{1}{1 + e^{-z}}.$$

The discrete cross-entropy function  $H(\mathbf{p}, \mathbf{q})$  measures the distance between the two distributions, where the probabilities are  $\mathbf{p} = [p_1, \dots, p_k]^T$  and  $\mathbf{q} = [q_1, \dots, q_k]^T$ .<sup>1</sup> The discrete cross-entropy is defined by

$$H(\mathbf{p}, \mathbf{q}) \triangleq - \sum_{i=1}^k p_i \log q_i.$$

Now we are ready to derive the logistic regression models.

**Logistic regression (variant 1).** The feature vector  $\mathbf{x} \in \mathbb{R}^d$  is the input to the model, and the label is  $y \in \{0, 1\}$  (different from the model (2.1)). The model has a trainable parameter vector

---

<sup>1</sup>Here,  $\mathbf{p}$  and  $\mathbf{q}$  are  $k$ -dimensional vectors such that  $p_1, \dots, p_k \geq 0$ ,  $q_1, \dots, q_k \geq 0$ ,  $p_1 + \dots + p_k = 1$ , and  $q_1 + \dots + q_k = 1$ .

$\mathbf{w} \in \mathbb{R}^d$ . The prediction is made by the sigmoid function of  $\mathbf{x}^T \mathbf{w}$ :

$$f(\mathbf{x}) = \text{sigmoid}(\mathbf{x}^T \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{x}^T \mathbf{w})}.$$

The function  $f(\mathbf{x}) \in (0, 1)$  is the predictor; you can think of it as a dense layer with sigmoid activation function. The function value is the probability that  $\mathbf{x}$  is a positive sample; therefore,  $1 - f(\mathbf{x})$  is the probability that  $\mathbf{x}$  is negative sample.

Next, we use the label  $y \in \{0, 1\}$  and the predictor  $f(\mathbf{x})$  to build a loss function. Let the ground truth be

$$\mathbf{p} = \begin{bmatrix} y \\ 1 - y \end{bmatrix}.$$

Let the predicted probability distribution be captured by

$$\mathbf{q} = \begin{bmatrix} f(\mathbf{x}) \\ 1 - f(\mathbf{x}) \end{bmatrix}.$$

Because  $\mathbf{p}$  and  $\mathbf{q}$  are all-positive vectors and  $p_1 + p_2 = 1$  and  $q_1 + q_2 = 1$ ,  $\mathbf{p}$  and  $\mathbf{q}$  capture two probability distributions. We hope the prediction  $\mathbf{q}$  is as close to the ground truth  $\mathbf{p}$  as possible. We use the cross-entropy function to measure the distance between  $\mathbf{p}$  and  $\mathbf{q}$ , and the resulting loss function is

$$\begin{aligned} \text{Loss}(\mathbf{x}, y) &= H(\mathbf{p}, \mathbf{q}) = -(p_1 \log q_1 + p_2 \log q_2) \\ &= -[y \log(f(\mathbf{x})) + (1 - y) \log(1 - f(\mathbf{x}))] \\ &= y \cdot \log [1 + \exp(-\mathbf{x}^T \mathbf{w})] + (1 - y) \mathbf{x}^T \mathbf{w} + (1 - y) \cdot \log [1 + \exp(-\mathbf{x}^T \mathbf{w})] \\ &= \log [1 + \exp(-\mathbf{x}^T \mathbf{w})] + (1 - y) \mathbf{x}^T \mathbf{w}. \end{aligned}$$

Using all the  $n$  training samples  $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ , the final logistic regression model is

$$\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n \text{Loss}(\mathbf{x}_i, y_i),$$

equivalently,

$$\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n \left\{ \log [1 + \exp(-\mathbf{x}_i^T \mathbf{w})] + (1 - y_i) \mathbf{x}_i^T \mathbf{w} \right\}. \quad (2.2)$$

We can add the  $\ell_1$  or  $\ell_2$ -norm regularization to the objective function.

**Logistic regression (variant 2).** This variant is the model (2.1) we study throughout. Different from the other variant, here we require the label  $y$  be either  $+1$  or  $-1$ . The model has a trainable parameter vector  $\mathbf{w} \in \mathbb{R}^d$ . We hope  $\text{sgn}(\mathbf{x}^T \mathbf{w})$  is the same as  $y$ ; equivalently, we hope  $y \mathbf{x}^T \mathbf{w}$  is always positive.

Based on such intuitions, we do the transformation for the training samples:

$$\tilde{\mathbf{x}} = y \mathbf{x}.$$

Therefore, we can always encourage  $\tilde{\mathbf{x}}^T \mathbf{w}$  to be positive. The prediction is made by the sigmoid function of  $\tilde{\mathbf{x}}^T \mathbf{w}$ :

$$f(\mathbf{x}, y) = \text{sigmoid}(\tilde{\mathbf{x}}^T \mathbf{w}) = \text{sigmoid}(y\mathbf{x}^T \mathbf{w}) = \frac{1}{1 + \exp(-y\mathbf{x}^T \mathbf{w})}.$$

The predictor  $f(\mathbf{w})$  is the probability that  $\tilde{\mathbf{x}}^T \mathbf{w}$  is positive; thus we encourage  $f(\mathbf{x})$  as close to 1 as possible. Let the ground truth be fixed as

$$\mathbf{p} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Let the prediction be

$$\mathbf{q} = \begin{bmatrix} f(\mathbf{x}, y) \\ 1 - f(\mathbf{x}, y) \end{bmatrix}.$$

We seek to make  $\mathbf{q}$  close to  $\mathbf{p}$  and thus define the loss function:

$$\begin{aligned} \text{Loss}(\mathbf{x}, y) &= H(\mathbf{p}, \mathbf{q}) = -(p_1 \log q_1 + p_2 \log q_2) = -p_1 \log q_1 \\ &= -\log(f(\mathbf{x}, y)) = \log[1 + \exp(-y\mathbf{x}^T \mathbf{w})]. \end{aligned}$$

Using all the  $n$  training samples  $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ , the final logistic regression model is

$$\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n \text{Loss}(\mathbf{x}_i, y_i).$$

If we further add the  $\ell_2$ -norm regularization, then the model becomes exactly (2.1).

**Remark.** Variant 1 is what we actually use in deep learning for binary classification, because the combination of sigmoid activation and binary cross-entropy is exactly the model (2.2). If we just do logistic regression itself, Variant 2 is easier to use than Variant 1.

### 3 Training (Numerical Optimization)

Given the training data  $\mathbf{X}$  and  $\mathbf{y}$ , we seek to solve the optimization model (2.1) to find the minimum  $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} Q(\mathbf{w}; \mathbf{X}, \mathbf{y})$ . Logistic regression does not have a close-formed solution, but we can use numerical optimization to approach  $\mathbf{w}^*$  up to any precision. In this section, we introduce gradient descent (GD), stochastic gradient descent (SGD), accelerated SGD, and mini-batch SGD; all of them can solve the logistic regression model (2.1).

#### 3.1 Gradients

Let  $u = -y_i \mathbf{w}^T \mathbf{x}_i$ ,  $v = e^u$ , and  $l_i = \log(1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)) = \log(1 + v)$ . We first compute the scalar derivatives:

$$\frac{\partial l_i}{\partial v} = \frac{\partial \log(1 + v)}{\partial v} = \frac{1}{1 + v} = \frac{1}{1 + e^u} \quad \text{and} \quad \frac{\partial v}{\partial u} = \frac{\partial e^u}{\partial u} = e^u.$$

We then compute the derivative of scalar w.r.t. vector:

$$\frac{\partial u}{\partial \mathbf{w}} = \frac{\partial (-y_i \mathbf{w}^T \mathbf{x}_i)}{\partial \mathbf{w}} = -y_i \mathbf{x}_i.$$

We finally apply the chain rule to get

$$\frac{\partial l_i}{\partial \mathbf{w}} = \frac{\partial u}{\partial \mathbf{w}} \frac{\partial v}{\partial u} \frac{\partial l_i}{\partial v} = -y_i \mathbf{x}_i \cdot e^u \cdot \frac{1}{1 + e^u} = \frac{-y_i \mathbf{x}_i}{1 + e^{-u}} = \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}^T \mathbf{x}_i)}. \quad (3.1)$$

We can apply (2.1) and (3.1) to calculate the gradient of  $Q$  (aka derivative of  $Q(\mathbf{w}; \mathbf{X}, \mathbf{y})$  w.r.t.  $\mathbf{w}$ ):

$$\nabla Q(\mathbf{w}) \triangleq \frac{\partial Q(\mathbf{w}; \mathbf{X}, \mathbf{y})}{\partial \mathbf{w}} = \frac{1}{n} \sum_{i=1}^n \frac{\partial l_i}{\partial \mathbf{w}} + \lambda \mathbf{w} = \frac{1}{n} \sum_{i=1}^n \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}^T \mathbf{x}_i)} + \lambda \mathbf{w}. \quad (3.2)$$

The following Python code calculates the full gradient of  $Q$  at  $\mathbf{w}$ . The inputs are  $\mathbf{w}$ ,  $\mathbf{X}$ ,  $\mathbf{y}$ , and  $\lambda$ ; the code returns the full gradient.

```

1 # Calculate the gradient
2 # Inputs:
3 #   w: d-by-1 matrix
4 #   x: n-by-d matrix
5 #   y: n-by-1 matrix
6 #   lam: scalar, the regularization parameter
7 # Return:
8 #   g: d-by-1 matrix, full gradient
9 def gradient(w, x, y, lam):
10     n, d = x.shape
11     yx = numpy.multiply(y, x) # n-by-d matrix
12     yxw = numpy.dot(yx, w) # n-by-1 matrix
13     vec1 = numpy.exp(yxw) # n-by-1 matrix
14     vec2 = numpy.divide(yx, 1+vec1) # n-by-d matrix
15     vec3 = -numpy.mean(vec2, axis=0).reshape(d, 1) # d-by-1 matrix
16     g = vec3 + lam * w
17     return g

```

### 3.2 Gradient descent (GD)

Let  $\mathbf{w}_0 \in \mathbb{R}^d$  be the initialization of  $\mathbf{w}$ ; for simple linear models, zero initialization is typically used. The gradient of  $Q$  at  $\mathbf{w}_0$ , denote

$$\nabla Q(\mathbf{w}_0) = \frac{1}{n} \sum_{i=1}^n \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}_0^T \mathbf{x}_i)} + \lambda \mathbf{w}_0,$$

is the steepest ascent direction at  $\mathbf{w}_0$ . In other words, if we want to increase the objective function  $Q$  as much as possible, but we are only allowed to move a tiny step away from  $\mathbf{w}_0$ , then we shall move  $\mathbf{w}$  along the gradient direction. Since we want to minimize the objective function  $Q$ , we move  $\mathbf{w}$  in the opposite direction of  $\nabla Q(\mathbf{w}_0)$ . So we update  $\mathbf{w}$  by:

$$\mathbf{w}_1 \longleftarrow \mathbf{w}_0 - \alpha \nabla Q(\mathbf{w}_0),$$

where  $\alpha$  is the step size (aka learning rate) and needs tuning. Then, we know  $\mathbf{w}_1$ , so we can calculate the gradient at  $\mathbf{w}_1$  using (3.2) to get

$$\nabla Q(\mathbf{w}_1) = \frac{1}{n} \sum_{i=1}^n \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}_1^T \mathbf{x}_i)} + \lambda \mathbf{w}_1.$$

We perform a similar update:

$$\mathbf{w}_2 \leftarrow \mathbf{w}_1 - \alpha \nabla Q(\mathbf{w}_1).$$

We can repeat this procedure so on and so forth till the gradient  $\nabla Q(\mathbf{w}_t)$  is close to zero or the objective function  $Q(\mathbf{w}; \mathbf{X}, \mathbf{y})$  stops decreasing.

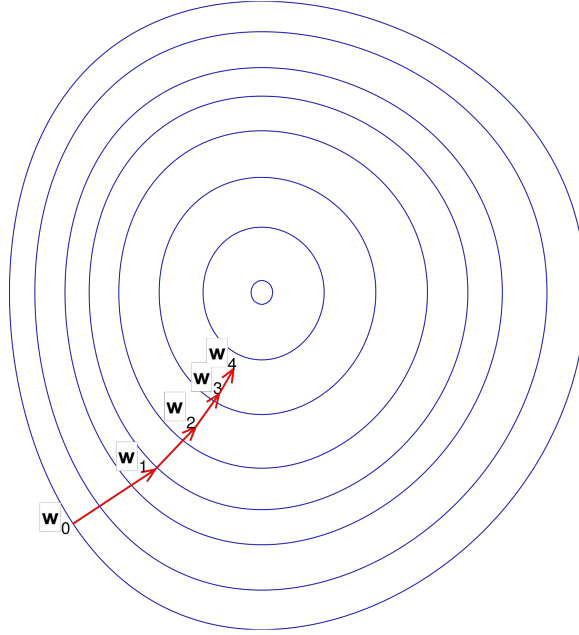


Figure 1: Gradient descent. The figure is from Wikipedia.

**Algorithm.** The gradient descent algorithm is illustrate in Figure 1 and formally described in the following.

1. Initialize  $\mathbf{w}_0$ , e.g.,  $\mathbf{w}_0 = \mathbf{0}$ ;
2. For  $t = 0, 1, 2, \dots, T$ , repeat the following steps:
  - (a) Use  $\mathbf{X}$ ,  $\mathbf{y}$ , and  $\mathbf{w}_t$  to calculate  $\nabla Q(\mathbf{w}_t) = \frac{1}{n} \sum_{i=1}^n \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}_t^T \mathbf{x}_i)} + \lambda \mathbf{w}_t$ ;
  - (b) Update  $\mathbf{w}$  by  $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla Q(\mathbf{w}_t)$ ;
3. Output  $\mathbf{w}_T$  as a (near) optimal solution to (2.1).

The following Python code implements the full gradient descent algorithm. The user needs to specify the step size (aka learning rate) and the maximal number of iterations (equal to epochs). The default initialization of  $\mathbf{w}$  is all-zero; however, if the user happens to have a good estimate of  $\mathbf{w}$ , then a warm start (using the estimate to initialize  $\mathbf{w}$ ) will make the convergence faster.

```

1 # Gradient descent for solving logistic regression
2 # Inputs:
3 #     x: n-by-d matrix
4 #     y: n-by-1 matrix
5 #     lam: scalar, the regularization parameter
6 #     stepsize: scalar
7 #     max_iter: integer, the maximal iterations
8 #     w: d-by-1 matrix, initialization of w
9 # Return:
10 #     w: d-by-1 matrix, the solution
11 #     objvals: a record of each iteration's objective value
12 def grad_descent(x, y, lam, stepsize, max_iter=100, w=None):
13     n, d = x.shape
14     objvals = numpy.zeros(max_iter) # store the objective values
15     if w is None:
16         w = numpy.zeros((d, 1)) # zero initialization
17
18     for t in range(max_iter):
19         objval = objective(w, x, y, lam)
20         objvals[t] = objval
21         print('Objective value at t=' + str(t) + ' is ' + str(objval))
22         g = gradient(w, x, y, lam)
23         w -= stepsize * g
24
25     return w, objvals
26
27 # example
28 lam = 1E-6
29 stepsize = 1.0
30 w, objvals_gd = grad_descent(x_train, y_train, lam, stepsize)

```

**Time complexity.** For the GD algorithm, each iteration (one update of  $\mathbf{w}$ ) goes one pass over all the  $n$  training samples. Thus one iteration equals one epoch. (Please refer to Section 3.7 for the definition of terminologies.) Each iteration of GD computes a full gradient, which costs  $\mathcal{O}(nd)$  time.

Let  $\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} Q(\mathbf{w}; \mathbf{X}, \mathbf{y})$  be the global minimum; for  $\lambda > 0$ , the global minimum is unique. Theories guarantees that with  $\lambda$  positive and  $\alpha$  set properly,  $\mathbf{w}_t$  converges to the global minimum  $\mathbf{w}^*$ . That is, for a sufficiently large  $T$  (may be several hundreds or thousands),  $\|\mathbf{w}_T - \mathbf{w}^*\|_2$  is sufficiently small.

### 3.3 Stochastic gradient descent (SGD)

In each iteration, the gradient descent algorithm computes  $\nabla Q(\mathbf{w}_t)$  according to (3.2). The per-iteration time complexity is  $\mathcal{O}(nd)$ . In fact, computing the full gradient is unnecessary at all; we can use stochastic gradient in lieu of the full gradient.



To derive stochastic gradient, let us express  $Q$  as

$$Q(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^n Q_i(\mathbf{w}), \quad (3.3)$$

where

$$Q_i(\mathbf{w}) = \log \left( 1 + \exp \left( -y_i \mathbf{w}^T \mathbf{x}_i \right) \right) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2.$$

Following Section 3.1, we can easily show that

$$\nabla Q_i(\mathbf{w}) \triangleq \frac{\partial Q_i(\mathbf{w})}{\partial \mathbf{w}} = \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}^T \mathbf{x}_i)} + \lambda \mathbf{w}. \quad (3.4)$$

**Algorithm.** Now we can present the SGD algorithm. SGD is almost the same to gradient descent except for using  $\nabla Q_i(\mathbf{w}_t)$  in lieu of  $\nabla Q(\mathbf{w}_t)$ . SGD is formally described in the following.

1. Initialize  $\mathbf{w}_0$ , e.g.,  $\mathbf{w}_0 = \mathbf{0}$ ;
2. For  $t = 0, 1, 2, \dots, T$ , repeat the following steps:
  - (a) Sample an index  $i$  from  $\{1, 2, \dots, n\}$  uniformly at random;
  - (b) Use  $\mathbf{x}_i$ ,  $y_i$ , and  $\mathbf{w}_t$  to calculate  $\nabla Q_i(\mathbf{w}_t) = \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}_t^T \mathbf{x}_i)} + \lambda \mathbf{w}_t$ ;
  - (c) Update  $\mathbf{w}$  by  $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla Q_i(\mathbf{w}_t)$ ;
  - (d) Decrease the step size  $\alpha$ ;
3. Output  $\mathbf{w}_T$  as a (near) optimal solution to (2.1).

The following Python code implements SGD for solving logistic regression. The function `stochastic_objective_gradient` calculates the value and gradient of  $Q_i$  at  $\mathbf{w}$ . The user needs to specify the step size (aka learning rate) and the maximal number of epochs. Different from GD, the learning rate of SGD should be decreased after every epoch to guarantee convergence (see Line 32).

The actual implementation of SGD is slightly different from the above pseudo-code. In each iteration of the pseudo-code, an index  $i$  is randomly sampled from the set  $\{1, \dots, n\}$ . In practice, a better option is in each epoch, randomly shuffling the  $n$  samples and then visiting the samples sequentially.

```

1 # Calculate the objective Q_i and the gradient of Q_i
2 # Inputs:
3 #     w: d-by-1 matrix
4 #     xi: 1-by-d matrix
5 #     yi: scalar
6 #     lam: scalar, the regularization parameter
7 # Return:
8 #     obj: scalar, the objective Q_i
9 #     g: d-by-1 matrix, gradient of Q_i
10 def stochastic_objective_gradient(w, xi, yi, lam):
11     d = xi.shape[0]
12     yx = yi * xi # 1-by-d matrix
13     yxw = float(numpy.dot(yx, w)) # scalar

```

```

14
15 # calculate objective function Q_i
16 loss = numpy.log(1 + numpy.exp(-yxw)) # scalar
17 reg = lam / 2 * numpy.sum(w * w) # scalar
18 obj = loss + reg
19
20 # calculate stochastic gradient
21 g_loss = -yx.T / (1 + numpy.exp(yxw)) # d-by-1 matrix
22 g = g_loss + lam * w # d-by-1 matrix
23
24 return obj, g

```

```

1 # SGD for solving logistic regression
2 # Inputs:
3 #     x: n-by-d matrix
4 #     y: n-by-1 matrix
5 #     lam: scalar, the regularization parameter
6 #     stepsize: scalar
7 #     max_epoch: integer, the maximal epochs
8 #     w: d-by-1 matrix, initialization of w
9 # Return:
10 #     w: the solution
11 #     objvals: record of each iteration's objective value
12 def sgd(x, y, lam, stepsize, max_epoch=100, w=None):
13     n, d = x.shape
14     objvals = numpy.zeros(max_epoch) # store the objective values
15     if w is None:
16         w = numpy.zeros((d, 1)) # zero initialization
17
18     for t in range(max_epoch):
19         # randomly shuffle the samples
20         rand_indices = numpy.random.permutation(n)
21         x_rand = x[rand_indices, :]
22         y_rand = y[rand_indices, :]
23
24         objval = 0 # accumulate the objective values
25         for i in range(n):
26             xi = x_rand[i, :] # 1-by-d matrix
27             yi = float(y_rand[i, :]) # scalar
28             obj, g = stochastic_objective_gradient(w, xi, yi, lam)
29             objval += obj
30             w -= stepsize * g
31
32         stepsize *= 0.9 # decrease step size; to be tuned
33         objval /= n
34         objvals[t] = objval
35         print('Objective value at epoch t=' + str(t) + ' is ' + str(objval))
36
37     return w, objvals
38
39 # example
40 lam = 1E-6
41 stepsize = 0.1
42 w, objvals_sgd = sgd(x_train, y_train, lam, stepsize)

```

**Time complexity.** Each iteration of SGD uses merely one training sample  $(\mathbf{x}_i, y_i)$  and costs  $\mathcal{O}(d)$  time. The per-epoch time complexity of SGD and GD are the same: both are  $\mathcal{O}(nd)$ . If we plot objective function against **iterations**, then GD will obviously look better than SGD. However, using **iterations** as the  $x$ -axis is very unfair. If we instead plot objective function against **epochs**, then SGD typically looks faster than GD, at least in the beginning.

**Theory.** Why does SGD work? It is because stochastic gradient is an unbiased estimate of the full gradient:

$$\mathbb{E}_i[\nabla Q_i(\mathbf{w})] = \sum_{j=1}^n \frac{1}{n} \nabla Q_j(\mathbf{w}) = \nabla Q(\mathbf{w}).$$

Here, the expectation is taken w.r.t. the random index  $i$ . Since  $i$  can be any of  $1, 2, \dots, n$  with probability  $\frac{1}{n}$ , the expectation can be written as a summation, by which the former identity follows. The latter identity follows from (3.3).

### 3.4 Mini-batch SGD

Mini-batch SGD (MB-SGD) is a straightforward extension of SGD. In the previous subsection, we defined the function

$$Q_i(\mathbf{w}) = \log(1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

and derived its gradient at  $\mathbf{w}$ ,  $\nabla Q_i(\mathbf{w})$ , as defined in (3.4). We showed  $\nabla Q_i(\mathbf{w})$  is an unbiased estimate of  $\nabla Q(\mathbf{w})$ , provided that  $i$  is sampled uniformly at random:

$$\mathbb{E}_{i \sim p}[\nabla Q_i(\mathbf{w})] = \nabla Q(\mathbf{w}), \quad \text{where } p(i) = \frac{1}{n} \text{ for } i \in \{1, \dots, n\}.$$

Let  $\mathcal{I}$  be a random set with cardinality  $b$ ; to be specific, every element in  $\mathcal{I}$  is uniformly sampled from  $\{1, \dots, n\}$  without replacement. Define the function

$$Q_{\mathcal{I}}(\mathbf{w}) = \frac{1}{b} \sum_{i \in \mathcal{I}} \log(1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2.$$

Its gradient at  $\mathbf{w}$  is

$$\nabla Q_{\mathcal{I}}(\mathbf{w}) \triangleq \frac{\partial Q_{\mathcal{I}}(\mathbf{w})}{\partial \mathbf{w}} = \frac{1}{b} \sum_{i \in \mathcal{I}} \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}^T \mathbf{x}_i)} + \lambda \mathbf{w}. \quad (3.5)$$

Mini-batch SGD uses  $\nabla Q_{\mathcal{I}}(\mathbf{w})$ , instead of  $\nabla Q_i(\mathbf{w})$ , to update  $\mathbf{w}$ . SGD is a special case of mini-batch SGD with  $b = 1$ . We can similarly prove that

$$\mathbb{E}_{\mathcal{I}}[\nabla Q_{\mathcal{I}}(\mathbf{w})] = \nabla Q(\mathbf{w}),$$

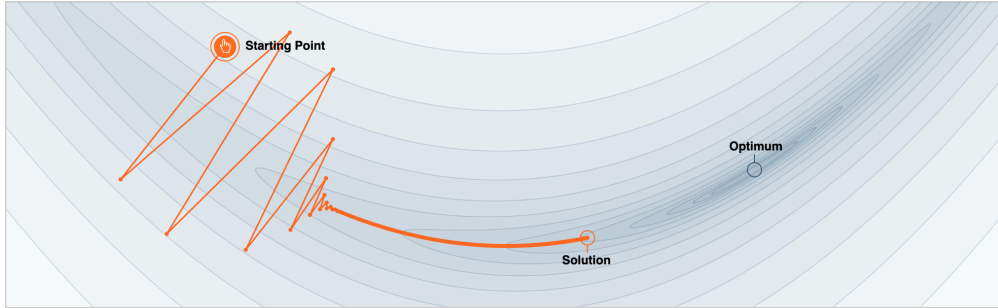
where the expectation is taken w.r.t. the random set  $\mathcal{I}$ .

**Algorithm.** The MB-SGD algorithm is almost the same as SGD except for using  $\nabla Q_{\mathcal{I}}(\mathbf{w}_t)$  instead of  $\nabla Q_i(\mathbf{w}_t)$ . MB-SGD is formally described in the following.

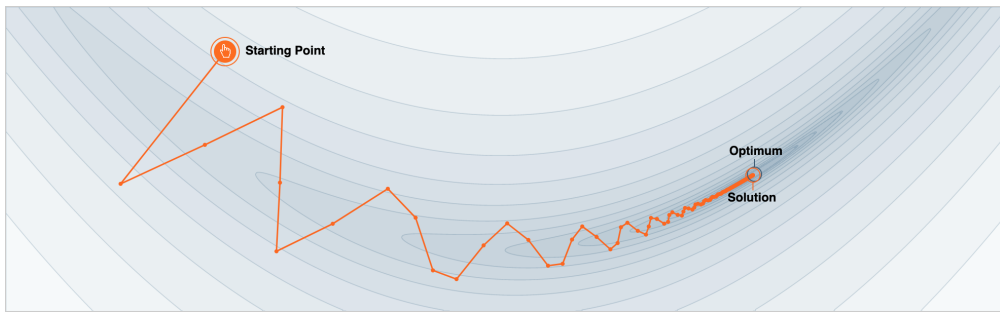
1. Initialize  $\mathbf{w}_0$ , e.g.,  $\mathbf{w}_0 = \mathbf{0}$ ;
2. For  $t = 0, 1, 2, \dots, T$ , repeat the following steps:
  - (a) Uniformly sample  $b$  indices from  $\{1, 2, \dots, n\}$  without replacement as a set  $\mathcal{I}$ ;
  - (b) Use  $\{(\mathbf{x}_i, y_i)\}_{i \in \mathcal{I}}$  and  $\mathbf{w}_t$  to calculate  $\nabla Q_{\mathcal{I}}(\mathbf{w}_t) = \frac{1}{b} \sum_{i \in \mathcal{I}} \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}_t^T \mathbf{x}_i)} + \lambda \mathbf{w}_t$ ;
  - (c) Update  $\mathbf{w}$  by  $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla Q_{\mathcal{I}}(\mathbf{w}_t)$ ;
  - (d) Decrease the step size  $\alpha$ ;
3. Output  $\mathbf{w}_T$  as a (near) optimal solution to (2.1).

MB-SGD can be implemented in Python by slightly changing the code of SGD.

MB-SGD has almost the same convergence properties as SGD. In practice, mini-batch is a default setting for SGD and other stochastic optimization algorithms, e.g., Adagrad, RMSProp, etc. When mentioning SGD, Adagrad, and RMSProp in practical papers and projects, people typically mean the mini-batch variants.



(a) SGD



(b) SGD with momentum

Figure 2: SGD and SGD with momentum. The figures are from <https://distill.pub/2017/momentum/>.

### 3.5 Accelerated GD and SGD

The convergence of GD and SGD critically depends on the **condition number** of the problem. For ill-conditioned problems (i.e.,  $\kappa$  is large), as illustrated in Figure 2, the convergence of SGD is extremely slow. The steepest descending direction is not necessarily the best descending direction; in fact, following the steepest descending direction, the path will be zig-zag.

Accelerated gradient algorithm uses the historical gradient directions to correct the current descending direction and circumvent the zig-zag like path. Accelerated gradient algorithm can significantly speed up computation, especially for ill-conditioned problems. There are many accelerated algorithms based on the same idea. Figure 2 shows the advantage of SGD with momentum.

Here we describe SGD with momentum; GD and MB-SGD with momentum are analogous. SGD with momentum maintains a momentum vector  $\mathbf{z}_t$  and take the momentum direction instead of the stochastic gradient direction. Here is the update rule:

$$\begin{aligned}\mathbf{z}_{t+1} &\leftarrow \beta \mathbf{z}_t + \nabla Q_i(\mathbf{w}_t), \\ \mathbf{w}_{t+1} &\leftarrow \mathbf{w}_t - \alpha \mathbf{z}_{t+1}.\end{aligned}$$

In the update rule,  $\alpha > 0$  and  $\beta \in (0, 1)$  are two hyper-parameters affecting the convergence rate. If the condition number is large (which is bad),  $\beta$  should be set large, e.g.,  $\beta = 0.99$ . For SGD with momentum, we can safely use fixed  $\alpha$ ; differently, for SGD,  $\alpha$  must decay after every epoch to guarantee convergence.

### 3.6 Fairly comparing algorithms

The following code compares GD and SGD by plotting their objective function values against epochs. The output of the code is Figure 3. Besides the plot of objective against epochs, you can also plot the training accuracy against epochs or the validation accuracy against epochs.

```
1 import matplotlib.pyplot as plt
2 %matplotlib inline
3
4 fig = plt.figure(figsize=(6, 4))
5
6 epochs_gd = range(len(objvals_gd))
7 epochs_sgd = range(len(objvals_sgd))
8
9 line0, = plt.plot(epochs_gd, objvals_gd, '—b', LineWidth=4)
10 line1, = plt.plot(epochs_sgd, objvals_sgd, '—r', LineWidth=2)
11 plt.xlabel('Epochs', FontSize=20)
12 plt.ylabel('Objective Value', FontSize=20)
13 plt.xticks(FontSize=16)
14 plt.yticks(FontSize=16)
15 plt.legend([line0, line1], ['GD', 'SGD'], fontsize=20)
16 plt.tight_layout()
17 plt.show()
18 fig.savefig('compare_gd_sgd.pdf', format='pdf', dpi=1200)
```

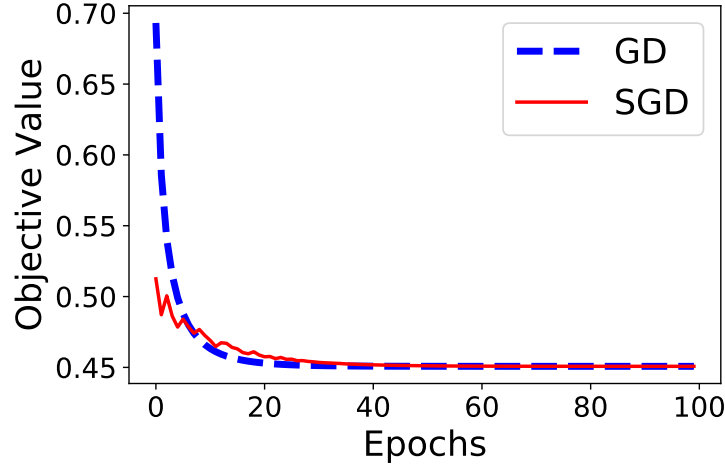


Figure 3: The plot of objective function value against epochs. SGD is tremendously faster than GD in the first 2 epochs, which means SGD can find a low-quality solution very efficiently.

The followings are typical pitfalls which the reader must avoid:

- Plot objective or accuracy against runtime. It is fine if the algorithms are implemented in Fortran, C, or C++. However, if your implementation is Python, MATLAB, or R, such a comparison is very unfair to SGD. It is because for-loop in Python, MATLAB, and R is very slow.
- Plot objective or accuracy against iterations. This is a very bad idea for comparing different algorithms. For example, in Figure 3, if you change  $x$ -axis to iterations, GD takes around 50 iterations to converge, while SGD takes around 20,000 iterations; however, such a comparison does not make any sense.

In sum, when you compare the convergence of two optimization algorithms, the most straightforward approach is plotting objective or accuracy against epochs.

### 3.7 Definitions of terminologies

**Iteration.** One iteration means performing one update of the optimization variable (aka trainable parameters)  $\mathbf{w}$ .

**Epoch.** One epoch means every of the  $n$  samples is visited exactly once by an algorithm. For GD, one epoch is equivalent to one iteration. For SGD, one epoch amounts to  $n$  iterations. For MB-SGD with batch size  $b$ , one epoch amounts to  $\frac{n}{b}$  iterations.

**Convergence.** An algorithm is said to converge to  $\mathbf{w}^*$  if  $\|\mathbf{w}_t - \mathbf{w}^*\|$  is arbitrarily close to zero for a sufficiently large  $T$  and all  $t \geq T$ .

**Warm-start.** Initialize  $\mathbf{w}$  using an estimate of  $\mathbf{w}^*$ . For example, with the regularization parameter  $\lambda = 10^{-3}$ , we obtain a solution  $\tilde{\mathbf{w}}$ ; then with a different setting,  $\lambda = 10^{-6}$ , we can use  $\tilde{\mathbf{w}}$  as the initialization.

**Strongly convex function.** The function's graph is above a convex quadratic function. Formally speaking, a differentiable function  $Q$  is  $\mu$ -strongly convex if

$$Q(\mathbf{v}) \geq Q(\mathbf{w}) + \nabla Q(\mathbf{w})^T(\mathbf{v} - \mathbf{w}) + \frac{\mu}{2}\|\mathbf{v} - \mathbf{w}\|_2^2,$$

for some  $\mu > 0$  and all  $\mathbf{w}$  and  $\mathbf{v}$ .

## 4 Making Predictions

By solving the logistic regression model (2.1), we can use the solution  $\mathbf{w} \in \mathbb{R}^d$  for making predictions. For a test sample  $\mathbf{x}' \in \mathbb{R}^d$ , we make prediction by

$$f(\mathbf{x}') = \text{sgn}(\mathbf{w}^T \mathbf{x}').$$

**Classification accuracy and error rate** are the commonly used evaluation metrics. For the class-balanced problems, that is, the numbers of positive samples and negative samples are comparable, we can safely use classification accuracy or error rate.

However, for the **class-imbalanced problems**, we shall never use such metrics. For example, suppose we want to apply logistic regression to detect phishing emails, and 0.1% among all the emails are phishing emails. Then the classifier  $f(\mathbf{x}') = -1$ , which always makes non-phishing prediction disregarding the email content, will achieve a classification accuracy of 99.9%. Such a classifier is obviously useless, although its classification accuracy is extremely high. For class-imbalanced problems, we can use any of **precision and recall**, **F-measure**, **receiver operating characteristic (ROC) curve**, or **area under the curve (AUC)** as the evaluation metric. Please refer to the Wikipedia pages for the definitions and examples.

## 5 Cross-Validation

In the logistic regression model (2.1), the optimization variable  $\mathbf{w}$  is called *parameters* or *weights* in machine learning. The parameters can be directly learned from the training data using numerical optimization.

**Hyper-parameters.** The other parameters, including the regularization parameter  $\lambda$ , learning rate  $\alpha$ , momentum parameter  $\beta$ , batch size  $b$ , and the number of epochs  $T$ , are called hyper-parameters. Their setting determines the parameter vector  $\mathbf{w}$ , which is why  $\lambda$ ,  $\alpha$ ,  $\beta$ ,  $b$ , and  $T$  are called hyper-parameters.

- Hyper-parameters in the model. In the logistic regression model (2.1),  $\lambda$  is the only hyper-parameter, and the optimal  $\mathbf{w}$  is a function of  $\lambda$ .

- Hyper-parameters in the optimization algorithm. The MB-SGD with momentum algorithm has 4 hyper-parameters: learning rate  $\alpha$ , momentum parameter  $\beta$ , batch size  $b$ , and the number of epochs  $T$ . Among them,  $\alpha$ ,  $\beta$ , and  $b$  affects the convergence rate and runtime, while  $T$  determines the closeness to the global minimum. For strongly convex models, the computed  $\mathbf{w}$  may not depend on these hyper-parameters, especially when  $T$  is sufficiently large.<sup>2</sup>

The hyper-parameters cannot be learned from the training data using numerical optimization. How can we determine the hyper-parameters?

**Cross-validation.** If our goal is a low classification error rate on the test set, then an intuitive idea would try all the possible combinations of hyper-parameters and adopt the combination which **minimizes the test error**. Unfortunately, this idea is very **wrong**. All the model parameters and hyper-parameters must be independent of the test data; it would be otherwise cheating. Keep in mind: Never use test data for hyper-parameter tuning!

The right approach to hyper-parameter tuning is **cross-validation**. Specifically, randomly partition the training data to two subsets: one is the training set, and the other is the validation set. Learn the model parameters on the training set, and then evaluate the classification error on the validation set; the error is called validation error. You can try different combinations of hyper-parameters and choose the combination which minimizes the validation error.

The simple idea works in practice. More often than not, the chosen combination of hyper-parameters may not be the “best”. Due to the randomness in the partition, the combination may happen to be the best for this specific partition; by doing another random partitioning, another combination will be selected.

**$k$ -fold cross-validation.** A better idea is to repeat the random partition many times to reduce variance, and it leads to the  $k$ -fold cross-validation. Figure 4 illustrates the 10-fold cross-validation. The training set is randomly partitioned to 10 subsets. The model is trained using 9 subsets and validated on the remaining subset; in this way, we get a validation error  $E_1$ . By doing the same for the other partitions, we will get the validation errors  $E_2, E_3, \dots, E_{10}$  and the mean  $E = \frac{1}{10}(E_1 + \dots + E_{10})$ . Notice that one combination of hyper-parameters leads to one validation error  $E$ ; we choose the combination of hyper-parameters (such as  $\lambda$  and  $\alpha$ ) that minimizes  $E$ .

Hyper-parameter tuning is computation intensive. Using the  $k$ -fold cross-validation, we have to train the model for  $k$  times, which leads to a significant increase in the computation. In addition, there may be many combinations of hyper-parameters, especially when the model and algorithm have many hyper-parameters. **If there are  $c$  combinations of hyper-parameters and we use the  $k$ -fold cross-validation, then we will have to solve the model for  $ck$  times!** Keep in mind, hyper-parameter tuning is the most time-consuming step in machine learning.

---

<sup>2</sup>For strongly convex models, the global minimum is unique, and the discussed algorithms all converge to the global minimum after sufficiently many epochs. However, for nonconvex models such as deep neural networks, the solution  $\mathbf{w}$  is a function of all the four hyper-parameters:  $\alpha$ ,  $\beta$ ,  $b$ , and  $T$ ; a small change of any of them will lead to a different solution  $\mathbf{w}$ .





Figure 4: Illustration of 10-fold cross-validation. The figure is from <http://karlrosaen.com/ml/learning-log/2016-06-20/>.

## 6 Problems

**P1.** Implement mini-batch stochastic gradient descent (MB-SGD) by following the implementation of SGD.

**P2.** Implement MB-SGD with momentum by slightly modifying the code of MB-SGD.

**P3.** Compare MB-SGD and MB-SGD with momentum by plotting objective value against epochs as in Figure 3.

**P4.** The  $d$ -dim vector  $\frac{\partial l_i}{\partial \mathbf{w}}$  is defined in (3.1). Calculate the derivative of vector w.r.t. vector:

$$\frac{\partial}{\partial \mathbf{w}} \left( \frac{\partial l_i}{\partial \mathbf{w}} \right) = \frac{\partial}{\partial \mathbf{w}} \left( \frac{-y_i \mathbf{x}_i}{1 + \exp(y_i \mathbf{w}^T x_i)} \right).$$

Notice that the result is a  $d \times d$  matrix.

**P5.** The Hessian matrix is the second derivative of the objective function  $Q(\mathbf{w}; \mathbf{X}, \mathbf{y})$  w.r.t.  $\mathbf{w}$ . Please derive the Hessian matrix (which is  $d \times d$ ):

$$\nabla^2 Q(\mathbf{w}) \triangleq \frac{\partial Q(\mathbf{w}; \mathbf{X}, \mathbf{y})}{\partial \mathbf{w} \partial \mathbf{w}^T} = \frac{1}{n} \sum_{i=1}^n \frac{\partial}{\partial \mathbf{w}} \left( \frac{\partial l_i}{\partial \mathbf{w}} \right) + \lambda \cdot \frac{\partial \mathbf{w}}{\partial \mathbf{w}}.$$

**P6.** For the regularized logistic regression model (2.1),  $\lambda$  is the only hyper-parameter needs fine tuning.<sup>3</sup> Implement 5-fold cross-validation to find out the “best”  $\lambda$ .

<sup>3</sup>If you use MB-SGD with momentum, a rough setting of the hyper-parameters  $\alpha$ ,  $\beta$ ,  $b$ , and  $T$ , will work well.

Hint: Select  $\lambda$  from a set such as  $\{10^{-10}, 10^{-9}, \dots, 10^0, 10^1, 10^2\}$  to minimize the validation error  $E = \frac{1}{5}(E_1 + \dots + E_5)$ .