Homework 2

201900800302 赵淇涛

1. Linear regression

Overview

Linear regression is known as a powerful method to fit a set of data points with a straight line.

In this part, data points are distributed following the two-dimensional Gaussian distribution. We would construct a weight matrix for a target straight line and adopt two strategies to optimize this matrix to well fit data points.

The first strategy is to directly calculate the closed-form solution of the weight matrix, since the linear regression is actually an optimization problem and luckily it has a closed-form solution from mathematical analysis. The second strategy achieves an optimal solution by gradient descent. **MSE** loss was used as an optimization target. At first I randomly set up a weight matrix and did gradient descent based on the gradient of MSE loss w.r.t. each element in the weight matrix until the optimal solution was reached.

Both two strategies could attain the same satisfying solution.

Code and process

0. Set up

```
import numpy as np
import matplotlib.pyplot as plt
```

1. Function *normal_equations* to compute the closed-form solution

```
def normal_equations(X, y):
    W = np.zeros((2, ))
    W = np.linalg.inv(X.T.dot(X)).dot(X.T.dot(y))
    plt.subplot(2, 1, 1)
    plt.scatter(X[:, 1], y, c='r')
    plt.plot(X[:, 1], X[:, 1] * W[1] + W[0], color="dodgerblue",
label="Normal_equations")
    plt.title('Linear regression with normal equations')
    plt.legend(loc="best", ncol=4)
    return W
```

2. Function *GD* to implement gradient descent

```
def GD(X, y, learning_rate, print_every=500):
    W = np.random.rand(2)
    t = 0
    loss = 0.5 * ((y - X.dot(W)) ** 2)
```

```
    loss = loss.mean()
    loss_history = [loss]

    while True:
        if t % print_every == 0:
            print("Iteration: %d, loss: %.4f" % (t, loss))
        grad = -((y - X.dot(W)) * (X.T)).T
        grad = grad.mean(axis=0)
        W -= learning_rate * grad
        loss = 0.5 * ((y - X.dot(W)) ** 2)
        loss = loss.mean()
        if abs(loss - loss_history[-1]) < 1e-6:
            print("Iteration: %d, loss: %.4f" % (t, loss))
            break
        loss_history.append(loss)
        t += 1

    plt.subplot(2, 1, 2)
    plt.scatter(X[:, 1], y, c='r')
    plt.plot(X[:, 1], X[:, 1] * W[1] + W[0], color="darkorange", label="GD")
    plt.title('Linear regression with gradient descent')
    plt.legend(loc="best", ncol=4)
    return W
```

3. Main part

```
mean = (1,2)
cov = [[1, -0.3], [-0.3, 2]]
dot_num = 30
learning_rate = 1e-3
np.random.seed(231)
dots = np.random.multivariate_normal(mean, cov, dot_num)
X = np.ones((dot_num, 2))
X[:, 1], y = dots[:, 0], dots[:, 1]

normal_equations(X, y)
GD(X, y, 5e-3)
plt.subplots_adjust(hspace=0.5)
plt.show()
```

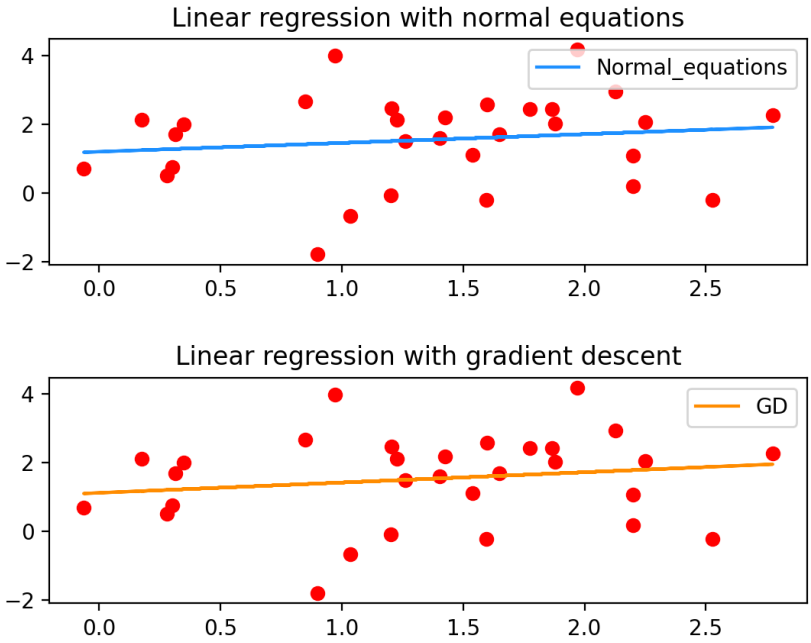
Result and discussion

1. Loss history

```
Iteration: 0, loss: 1.2050
Iteration: 500, loss: 0.8513
Iteration: 1000, loss: 0.8434
Iteration: 1500, loss: 0.8399
Iteration: 2000, loss: 0.8384
Iteration: 2386, loss: 0.8379
```

In the process of gradient descent, it took 2386 iterations to converge. Note that the learning_rate was 1e-3, a relatively conservative value selected by habit.

2. Result plot



Two strategies yielded exactly the same solution.

2. Logistic regression

Overview

While **linear regression** fits data points by a straight line, **logistic regression** uses a straight line as well to classify data points.

Given two sets of points obeying two-dimensional Gaussian distributions centered at distinct points respectively, a logistic function was defined to produce the probabilities that a specific point belongs to each of two data sets. Here, a weight matrix was also set up, by which coordinates of data points (plus an intercept term) were multiplied. Since inputs to the logistic function were generated by matrix multiplication, the decision boundary in this problem is linear.

We have no closed-form solution in this case. Hence, **s**tochastic **g**radient **d**escent (**SGD**) and ordinary **g**radient **d**escent (**GD**) were implemented to maximize the likelihood that a point was correctly classified. (In fact, in terms of maximization, they are more precisely the gradient ascent.)

Code and process

0. Set up

```
import numpy as np
import matplotlib.pyplot as plt
```

1. Function *generate_dots* to produce data points

```
def generate_dots(dot_num):
    mean_A = (0, 0)
    cov_A = [[1, 0], [0, 1]]
    mean_B = (1, 2)
    cov_B = [[1, -0.3], [-0.3, 2]]
    A = np.random.multivariate_normal(mean_A, cov_A, dot_num)
    B = np.random.multivariate_normal(mean_B, cov_B, dot_num)
    X_A = np.ones((dot_num, 3))
    X_B = np.ones((dot_num, 3))
    X_A[:, 1:3], X_B[:, 1:3] = A, B
    X = np.concatenate((X_A, X_B), axis=0)
    return X
```

2. *Logistic* function

```
def logistic(W, X):
    return 1 / (1+np.exp(X.dot(W)))
```

3. Function *GD* to implement **GD** or **SGD**

```
def GD(X, y, mode, learning_rate, print_every=200):
    N, D = X.shape
    W = np.random.rand(3)
    t = 0
    log_likelihood = y * (X.dot(W)) - np.log(1 + np.exp(X.dot(W)))
    log_likelihood = log_likelihood.sum()
    log_likelihood_history = [log_likelihood]
    while True:
        if t % print_every == 0:
            print("Epoch: %d, log_likelihood: %.4f" % (t, log_likelihood))
        grad = y - np.exp(X.dot(W))/(1 + np.exp(X.dot(W)))
        grad = ((X.T * grad).T)
        if mode == 'SGD':
            grad = grad[np.random.randint(N), :]
        else:
            grad = grad.mean(axis=0)
        W += learning_rate * grad
        log_likelihood = y * X.dot(W) - np.log(1 + np.exp(X.dot(W)))
        log_likelihood = log_likelihood.sum()
        if abs(log_likelihood - log_likelihood_history[-1]) < 1e-6:
            break
        log_likelihood_history.append(log_likelihood)
        t += 1
    print("Epoch: %d, log_likelihood: %.4f" % (t, log_likelihood))
    return W
```

4. Function *test* to test classifier

```
def test(W, mode):
    X = generate_dots(dot_num)
    y_A = np.zeros((dot_num, ))
    y_B = np.ones((dot_num, ))
    y = np.concatenate((y_A, y_B), axis=0)
    plt.scatter(X[:50, 1], X[:50, 2], c='b', label="A")
    plt.scatter(X[50:, 1], X[50:, 2], c='r', label="B")
    plt.plot(X[:, 1], (-W[1]*X[:, 1]-W[0])/W[2], color="darkorange",
label="Decision boundary")
    plt.title("Logistic regression with " + mode)
    plt.legend(loc="best", ncol=4)
    plt.show()
```

5. Main part

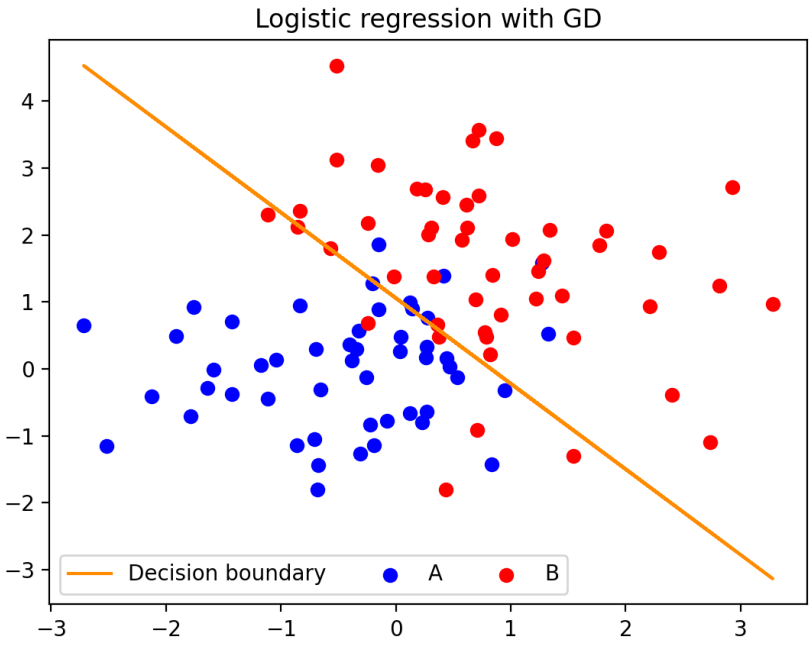
```
dot_num = 50
learning_rate = 5e-2
np.random.seed(231)
X = generate_dots(dot_num)
y_A = np.zeros((dot_num, ))
y_B = np.ones((dot_num, ))
y = np.concatenate((y_A, y_B), axis=0)

# Do 'GD' or 'SGD'
# W = GD(X, y, 'GD', learning_rate)
W = GD(X, y, 'SGD', learning_rate)
test(W, 'SGD')
```

Result and discussion

1. GD
    1. Loss history

```
Epoch: 0, log_likelihood: -51.8777
Epoch: 200, log_likelihood: -36.2312
Epoch: 400, log_likelihood: -33.9628
Epoch: 600, log_likelihood: -33.2388
Epoch: 800, log_likelihood: -32.9419
Epoch: 1000, log_likelihood: -32.8038
Epoch: 1200, log_likelihood: -32.7348
Epoch: 1400, log_likelihood: -32.6986
Epoch: 1600, log_likelihood: -32.6791
Epoch: 1800, log_likelihood: -32.6684
Epoch: 2000, log_likelihood: -32.6623
Epoch: 2200, log_likelihood: -32.6589
Epoch: 2400, log_likelihood: -32.6570
Epoch: 2600, log_likelihood: -32.6559
Epoch: 2800, log_likelihood: -32.6553
Epoch: 3000, log_likelihood: -32.6549
Epoch: 3121, log_likelihood: -32.6547
```

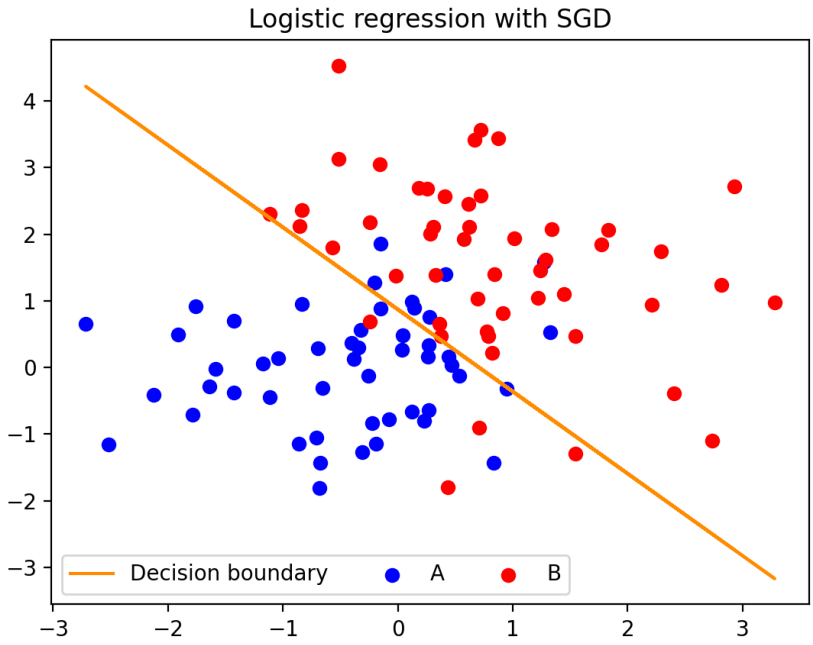
    2. Result plot



2. SGD
    1. Loss history

```
Epoch: 0, log_likelihood: -51.8777
Epoch: 200, log_likelihood: -36.1030
Epoch: 400, log_likelihood: -33.7032
Epoch: 600, log_likelihood: -33.2869
Epoch: 800, log_likelihood: -33.4161
Epoch: 1000, log_likelihood: -32.8495
Epoch: 1200, log_likelihood: -33.0468
Epoch: 1400, log_likelihood: -33.2630
Epoch: 1600, log_likelihood: -32.7374
Epoch: 1800, log_likelihood: -32.7031
Epoch: 2000, log_likelihood: -32.9923
Epoch: 2114, log_likelihood: -33.1453
```

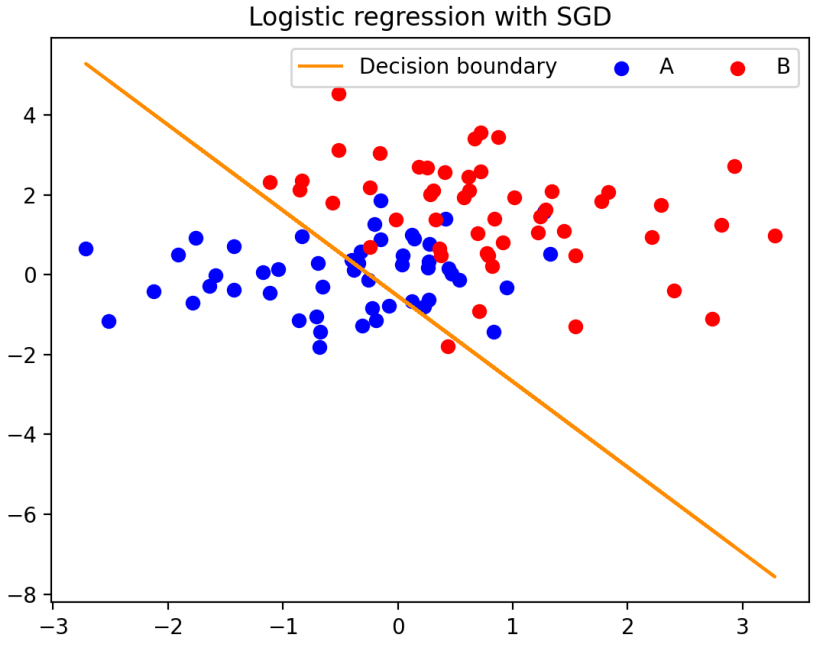
    2. Result plot



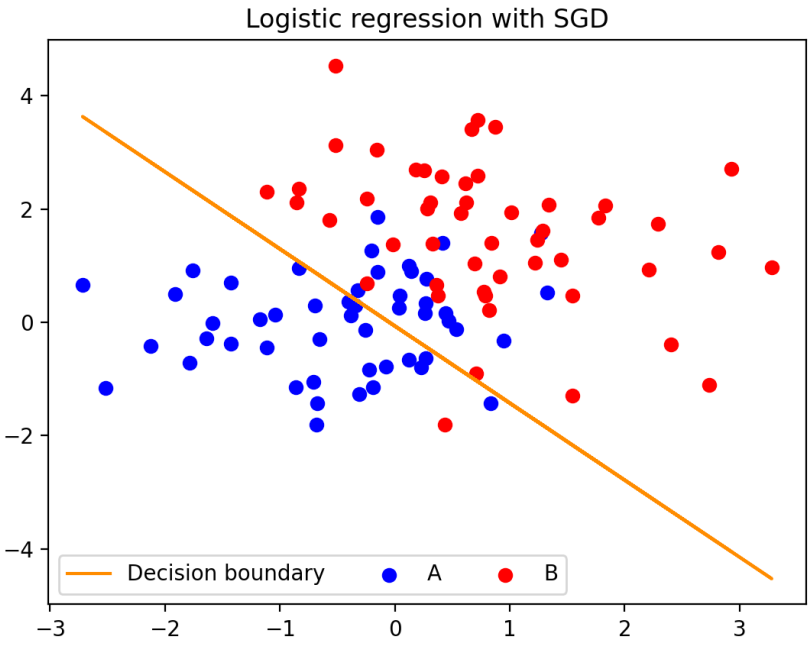
Note that both decision boundaries are almost identical, but SGD took far less steps to converge (2114 versus 3121).

Some plots of intermediate steps are as follows.

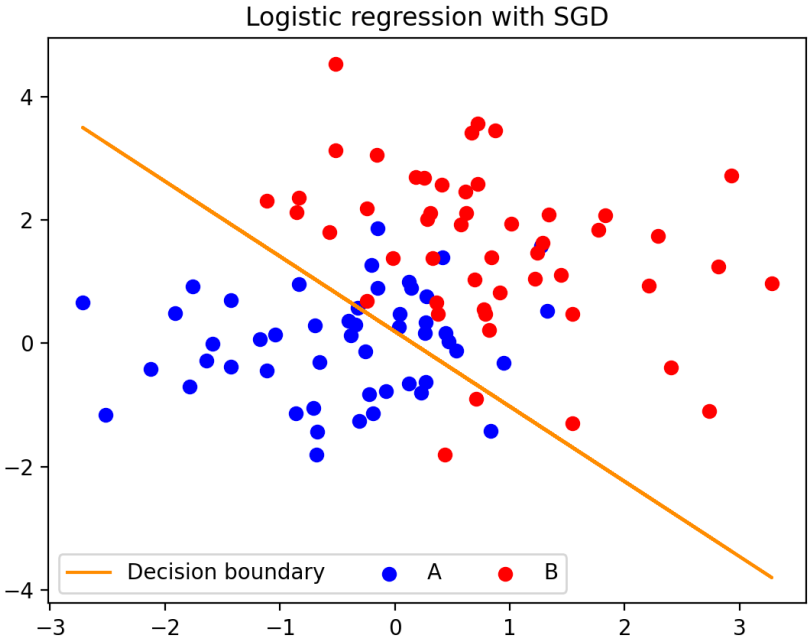
1. Step 10



2. Step 30



3. Step 50



The model converged fast in even first tens of steps.