Homework 3

201900800302 赵淇涛

1. Perceptron

- Question -

1. No. The order of examples affects the performance of the perceptron model.
2. Here two sets of indentical points that are differently ordered are given. Note that the intercept term of x is omitted and theta starts from (0, 0). The predicted label of data example is 1 if the dot production between X and theta is non-negtive.

   The first data set S:

x_0	x_1	y	theta_0	theta_1	Right/wrong
1	1	1	0	0	Right
1	-5	-1	0	0	Wrong
-1	1	-1	-1	5	Wrong
-2	1	-1	0	4	Wrong

   The second data set S':

x_0	x_1	y	theta_0	theta_1	Right/wrong
-1	1	-1	0	0	Wrong
-2	1	-1	1	-1	Right
1	-5	-1	1	-1	Wrong
1	1	1	0	4	Right

   S and S' yield different number of errors.

   Some data points of large value significantly change the value of theta, therefore their positions in the data set matter a lot.

   A long sequence of data points of the same label may not be good, which update the theta in a single direction and then keep it unchanged. Hence, a shuffling in training data samples is necessary.

- Experiment -

Overview

Perceptron is a simple but useful algorithm to implement classification.

In this part, data points are distributed following the two-dimensional Gaussian distribution. We would construct a weight matrix for a target straight line, serving as a decision boundary.

Given two sets of data points centered at distinct dots respectively and labeled as 1 or -1, a batch training algorithm is adopted. In each epoch, the weight matrix does dot production with each of the data points in the training set, the sign of which is exactly the predicted label provided input data points. If any error occurs, the weight matrix would be updated (plus the coordinate of the misclassified input data point multiplied by its correct label).

Code and process

0. Set up

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

1. Function *generate_data*

```
def generate_data(dot_num):
    mean_A = (0, 0)
    cov_A = [[1, 0], [0, 1]]
    mean_B = (1, 2)
    cov_B = [[1, 0], [0, 2]]
    X_A = np.random.multivariate_normal(mean_A, cov_A, dot_num)
    X_B = np.random.multivariate_normal(mean_B, cov_B, dot_num)
    X = np.ones((2*dot_num, 3))
    X[:, 1:] = np.concatenate((X_A, X_B), axis=0)
    y = np.zeros(2*dot_num)
    y[:dot_num] += 1
    y[dot_num:] -= 1
    return X, y
```

2. Function *train* to update the weight matrix

```
def train(X_train, y_train, epoch, learning_rate):
    N, D = X_train.shape
    theta = np.zeros(3)
    y_prediected = np.zeros_like(y_train)

    for i in range(epoch):

        scores = X_train.dot(theta)
        y_prediected[scores>=0] = 1
        y_prediected[scores<0] = -1
        correct = (y_prediected==y_train).sum()

        if i % 100 == 0:
            learning_rate /= 2
            print("Epoch: %d, accuracy: %.4f (%d/%d)" % (i, correct/N, correct, N))

        for j in range(N):
            temp = theta.dot(X_train[j, :])
            y = 1 if temp >=0 else -1
            if y != y_train[j]:
```

```
                theta += y_train[j] * X_train[j] * learning_rate

    print("Epoch: %d, accuracy: %.4f (%d/%d)" % (i, correct/N, correct, N))

    return theta
```

Here the *learning_rate* is introduced as in other ML tasks, which is divided by 2 every 100 epochs. However, empirically speaking, a constant *learning_rate* is equally good.

3. Function *test* to do classification

```
def test(X_test, y_test, theta):
    N, D = X_train.shape
    y_predicted = np.zeros(N)
    scores = X_test.dot(theta)
    y_predicted[scores>=0] = 1
    y_predicted[scores<0] = -1
    correct = (y_predicted == y_test).sum()
    print("Test accuracy: %.2f%% (%d/%d)" % (correct/N*100, correct, N))
```

4. Main part

```
np.random.seed(2021)

dot_num = 30
epoch = 10000
learning_rate = 2

# Generate data points
index = np.arange(2*dot_num)
np.random.shuffle(index)
X_train, y_train = generate_data(dot_num)
X_train, y_train = X_train[index, :], y_train[index]
X_test, y_test = generate_data(dot_num)
theta = train(X_train, y_train, epoch, learning_rate)
predicted_y = test(X_test, y_test, theta)

plt.scatter(X_test[:dot_num, 1], X_test[:dot_num, 2], c='b', label="A")
plt.scatter(X_test[dot_num:, 1], X_test[dot_num:, 2], c='r', label="B")
plt.plot(X_test[:, 1], -(X_test[:, 1]*theta[1]+theta[0])/theta[2], color="darkorange",
label="Decision boundary")
plt.title("Perceptron")
plt.legend(loc="best", ncol=4)
plt.show()
```

As shown above, the order of data points in the training set affects the performance of the model. Hence, a shuffling is operated. After trying several random seeds, one showing a preferable result is fixed.

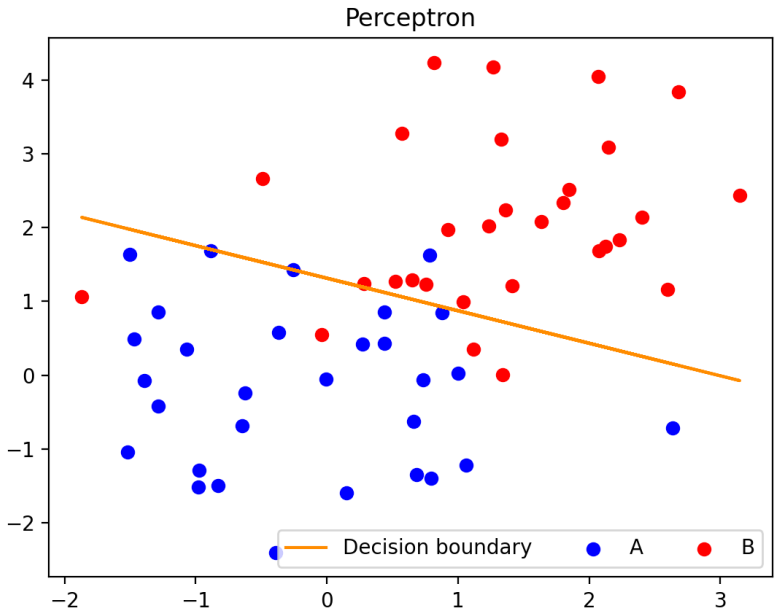
Result and discussion

1. Accuracy history

```
Epoch: 0, accuracy: 0.5000 (30/60)
Epoch: 1000, accuracy: 0.9000 (54/60)
Epoch: 2000, accuracy: 0.8500 (51/60)
Epoch: 3000, accuracy: 0.8500 (51/60)
Epoch: 4000, accuracy: 0.9000 (54/60)
Epoch: 5000, accuracy: 0.9000 (54/60)
Epoch: 6000, accuracy: 0.9333 (56/60)
Epoch: 7000, accuracy: 0.9500 (57/60)
Epoch: 8000, accuracy: 0.9167 (55/60)
Epoch: 9000, accuracy: 0.9500 (57/60)
Epoch: 9999, accuracy: 0.8833 (53/60)
Test accuracy: 91.67% (55/60)
```

Note that the training accuracy reached 90% within even first 1000 epochs and oscillated up and down afterwards.

2. Result plot



In the test process, another 60 points distributed the same as points in the training set were generated. The decision boundary is seemingly reasonable.

3. SVM

Overview

SVM was commonly used as a linear classifier before the prosperity of deep learning. With the kernel trick, SVM is equipped with non-linearity and can thus classify non-linearly.

In this part, given two sets of data point, data points in the first set follow the two-dimensional Gaussian distribution centered at origin and these in the second one are distributed uniformly over a rectangular region. A SVM classifier with kernel would be implemented.

Algorithm in detail

SVM was designed to solve an optimization problem below (It separates two sets of data points linearly with as large a margin as possible and tolerates errors to some extent at the same time):

$$\min_{\omega,b,\xi} \ \frac{1}{2}||\omega||^2 + C\sum_n \xi_n$$
$$subj.to \ \ y_n(\omega \cdot x_n + b) \geq 1 - \xi_n$$
$$\xi_n \geq 0$$

Using Lagrange multipliers, the problem can be written as:

$$\min_{\omega,b,\xi} \max_{\alpha \geq 0} \max_{\beta \geq 0} L(\omega, b, \xi, \alpha, \beta),$$

$$where \ L(\omega, b, \xi, \alpha, \beta) = \frac{1}{2}||\omega||^2 + C\sum_n \xi_n - \sum_n \beta_n \xi_n - \sum_n \alpha_n [y_n(\omega \cdot x_n + b) - 1 + \xi_n] Takeagradient$$

Take a gradient with respect to ω, set it equal to zero, and solve for ω in terms of other variables.

$$\bigtriangledown_\omega L = \omega - \sum_n \alpha_n y_n x_n = 0$$
$$\omega = \sum_n \alpha_n y_n x_n$$

It's quite similar to the expression of ω in perceptron.

Replace ω in the expression for L by this new expression and further calculate the gradient of L w.r.t. b, ξ_n. The L expression ends up being:

$$L(\alpha) = \sum_n \alpha_n - \frac{1}{2}\sum_n \sum_m \alpha_n \alpha_m y_n y_m x_n \cdot x_m$$

Note that $x_n \cdot x_m$ can be replaced by $K(x_n, x_m)$ using the kernel trick.

Removing variables adds a constraint to α, the final optimization problem becomes:

$$\min_\alpha -L(\alpha) = \frac{1}{2}\sum_n \sum_m \alpha_n \alpha_m y_n y_m K(x_n, x_m) - \sum_n \alpha_n$$
$$subj.to \ 0 \leq \alpha_n \leq C$$

Above expressions are given in the book *A Course in Machine Learning*. To implement gradient descent, the gradient of ith element of α w.r.t. $-L$ reads:

$$\bigtriangledown_{\alpha_i} -L = y_i \sum_n \alpha_n y_n K(x_n, x_i) - 1$$

To satisfy the costraint on α, after each gradient descent step, elements would be squashed into [0, C] if any violation happens.

The prediction function is written as $f(\widehat{x}) = sign(\sum_n \alpha_n y_n K(x_n, \widehat{x}))$, where x_n is each of training samples.

As for the choice of the form of the kernel, the Gaussian kernel is adopted.

$$K(x, z) = exp(-\frac{||x - z||_2^2}{2\sigma^2})$$

On the one hand, the Gaussian kernel has a strong non-linearity with infinite dimensional feature space. On the other, intuitively, given points in one set obey two-dimensional Gaussian distribution, the Gaussian kernel is expected to perform well.

Code and process

0. Set up

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

1. Function *generate_data* to produce data points

```
def generate_data(dot_num):
    mean = (0, 0)
    cov = [[1, 0], [0, 2]]
    X_A = np.random.multivariate_normal(mean, cov, dot_num)
    X_B = np.random.uniform([-5, -5], [5, 5], [dot_num, 2])
    X = np.concatenate((X_A, X_B), axis=0)
    y = np.zeros(2*dot_num)
    y[:dot_num] -= 1
    y[dot_num:] += 1
    return X, y
```

2. Function *linear_kernel* and *gaussian_kernel*

```
def linear_kernel(X_1, X_2):
    N1, D = X_1.shape
    if len(X_2.shape) == 1:
        N2 = 1
    else:
        N2, D = X_2.shape
    G = np.ones((N1, N2, D))
    G = G * X_2
    G = (G.transpose(1,0,2) * X_1).transpose(1, 0, 2)
    return G.sum(axis=2)

def gaussian_kernel(X_1, X_2):
    N1, D = X_1.shape
    if len(X_2.shape) == 1:
        N2 = 1
    else:
        N2, D = X_2.shape
    G = np.zeros((N1, N2, D))
    G = (G.transpose(1,0,2) + X_1).transpose(1, 0, 2)
    G -= X_2
    G = np.linalg.norm(G, axis=2) ** 2
    return np.exp(-G/(2 * sigma**2))
```

3. Function *rectifier* to ensure α is in the right range

```
def rectifier(x, C):
    return np.maximum(np.minimum(x, C), 0)
```

4. Function *train* to do gradient descent

```
def train(X_train, y_train, epoch, learning_rate):
    N, D = X_train.shape
    alpha = np.random.uniform(0, C, (N, ))
    G = gaussian_kernel(X_train, X_train)
    loss_history, accuracy_history = [], []

    for i in range(epoch):
        temp = (G * alpha * y_train).T * y_train
        temp = temp.T
        grad = temp.sum(axis=1) - 1
        loss = -alpha.sum() + (0.5 * temp.T * alpha).T.sum()
        loss_history.append(loss)

        if i % 100 == 0:
            y = np.zeros(N)
            for j in range(N):
                G_temp = gaussian_kernel(X_train, X_train[j, :])
                score = ((G_temp.T * alpha * y_train).T).sum(axis=0)
                y[j] = 1 if score >= 0 else -1
            correct = (y == y_train).sum()
            accuracy_history.append(correct/N)
            print("Epoch: %d, loss: %.4f, training accuracy: %.2f%% (%d/%d)" % (i, loss,
correct/N*100, correct, N))

        alpha -= learning_rate * grad
        alpha = rectifier(alpha, C)

    print("Epoch: %d, loss: %.4f, training accuracy: %.2f%% (%d/%d)" % (i, loss,
correct/N*100, correct, N))

    return alpha
```

5. Function *test* to test classifier

```
def test(X_train, y_train, X_test, y_test, alpha):
    N, D = X_train.shape
    y = np.zeros(N)
    for i in range(N):
        G = gaussian_kernel(X_train, X_test[i, :])
        score = ((G.T * alpha * y_train).T).sum()
        y[i] = 1 if score >= 0 else -1
    correct = (y == y_test).sum()
    print("Test accuracy: %.2f%% (%d/%d)" % (correct/N*100, correct, N))
    return y, correct/N
```

6. Function *func* to plot contours

```
def func(X, Y):
    scores = np.zeros_like(X)
    for i in range(100):
        for j in range(100):
            temp = np.array([X[i, j], Y[i, j]])
            G = gaussian_kernel(X_train, temp)
            scores[i, j] = ((G.T * alpha * y_train).T).sum()
    return scores
```

7. Main part

```
np.random.seed(2021)

dot_num = 100
C = 5.134395 # np.random.uniform(10)
epoch = 1000
alpha = np.random.uniform(0, C, (2*dot_num, ))
learning_rate = 1e-2
sigma = 0.82836 # 10 ** np.random.uniform(-2, 0)

# Generate data points
X_train, y_train = generate_data(dot_num)
X_test, y_test = generate_data(dot_num)

plt.figure(figsize=(8.5,10))

plt.subplot(3, 1, 1)
plt.scatter(X_test[:dot_num, 0], X_test[:dot_num, 1], s=15, c='b', label="A")
plt.scatter(X_test[dot_num:, 0], X_test[dot_num:, 1], s=15, c='r', label="B")
plt.title("Original distribution")
plt.legend(loc="best", ncol=4)

alpha_1000 = train(X_train, y_train, epoch, alpha, learning_rate)

predicted_y, accuracy_0 = test(X_train, y_train, X_test, y_test, alpha)
mask_A = np.where(predicted_y==-1)
mask_B = np.where(predicted_y==1)

predicted_y_1000, accuracy_1000 = test(X_train, y_train, X_test, y_test, alpha_1000)
mask_A_1000 = np.where(predicted_y_1000==-1)
mask_B_1000 = np.where(predicted_y_1000==1)

x = np.linspace(-5, 5, dot_num)
y = np.linspace(-5, 5, dot_num)
X, Y = np.meshgrid(x, y)

plt.subplot(3, 1, 2)
plt.contourf(X, Y, func(X, Y, alpha))
plt.scatter(X_test[mask_A, 0], X_test[mask_A, 1], s=15, c='b', label="A")
```

```
plt.scatter(X_test[mask_B, 0], X_test[mask_B, 1], s=15, c='r', label="B")
plt.title("Predicted distribution before training, test accuracy: " + str(100*accuracy_0) +
"%")
plt.legend(loc="best", ncol=4)

plt.subplot(3, 1, 3)
plt.contourf(X, Y, func(X, Y, alpha_1000))
plt.scatter(X_test[mask_A_1000, 0], X_test[mask_A_1000, 1], s=15, c='b', label="A")
plt.scatter(X_test[mask_B_1000, 0], X_test[mask_B_1000, 1], s=15, c='r', label="B")
plt.title("Predicted distribution after training, test accuracy: " + str(100*accuracy_1000)
+ "%")
plt.legend(loc="best", ncol=4)
plt.subplots_adjust(hspace=0.35, top=0.95)
plt.show()
```

Note that hyper-parameters are selected by random sampling.

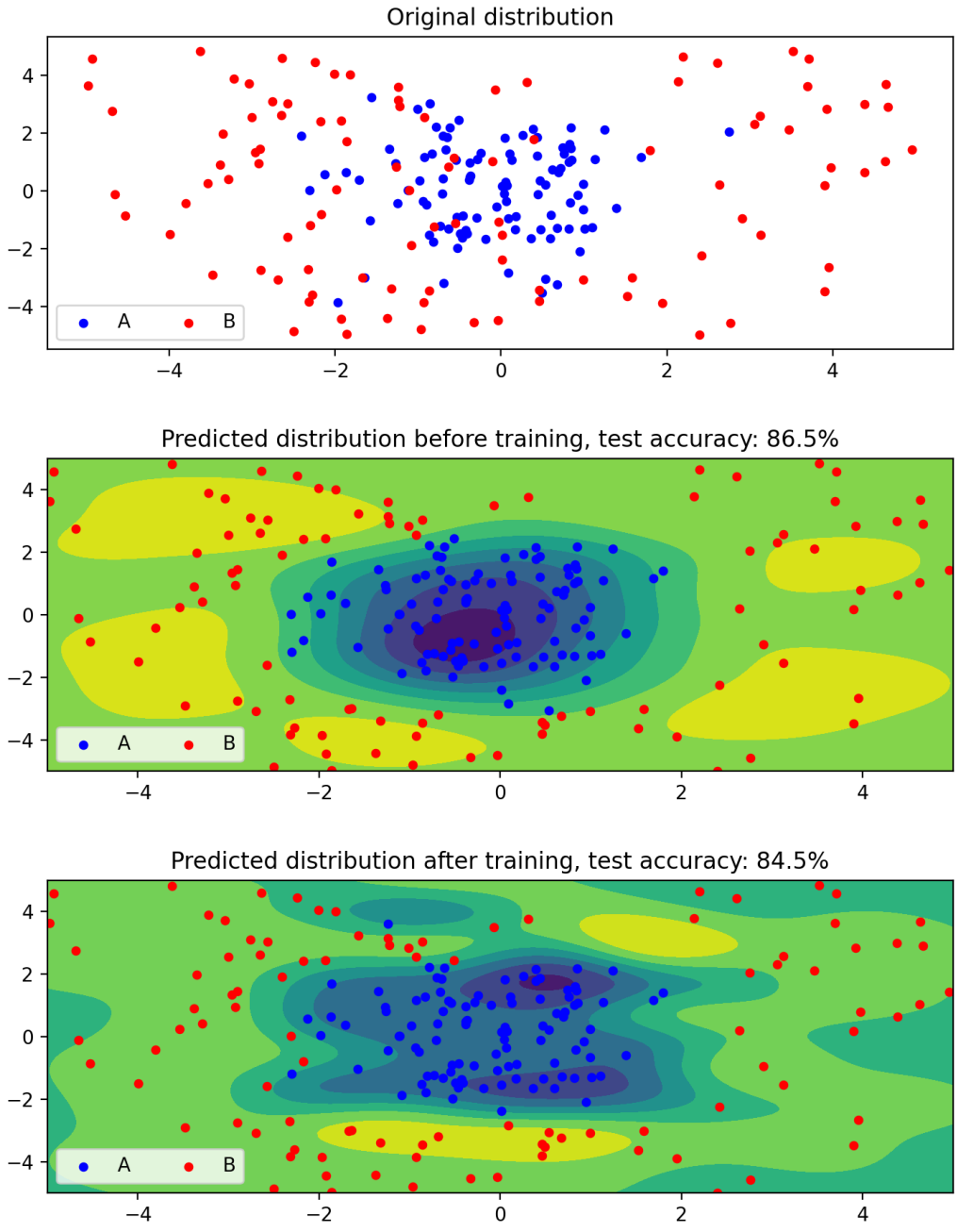
Result and discussion

1. Training process

```
Epoch: 0, loss: 5972.2265, training accuracy: 87.50% (175/200)
Epoch: 100, loss: -193.0719, training accuracy: 90.50% (181/200)
Epoch: 200, loss: -216.2989, training accuracy: 91.50% (183/200)
Epoch: 300, loss: -226.4444, training accuracy: 92.00% (184/200)
Epoch: 400, loss: -233.6102, training accuracy: 92.00% (184/200)
Epoch: 500, loss: -238.3543, training accuracy: 91.00% (182/200)
Epoch: 600, loss: -240.5136, training accuracy: 91.00% (182/200)
Epoch: 700, loss: -241.8471, training accuracy: 91.00% (182/200)
Epoch: 800, loss: -242.5945, training accuracy: 91.00% (182/200)
Epoch: 900, loss: -243.0597, training accuracy: 91.00% (182/200)
Epoch: 999, loss: -243.3565, training accuracy: 91.00% (182/200)
```

   The model converges fast. Amazingly, it achieves a good training accuracy before even training!

2. Result plot



The first plot shows the original distribution of test samples and the following two show the predicted results. Contours are plotted based on score at each point in the region, the sign of which determines the predicted label and blue indicates small values whereas yellow indicates high scores.

The Gaussian kernel reveals its capacity by correctly classify the blue data points centered at the origin.

It is worth noting that after training 1000 epochs, the test accuracy decreases. The reason is the distribution of data points is highly consistent with the choice of kernel, which embeds prior knowledge to some extent, so it performs well even before training. Training process makes model overfit the training samples (91% accuracy) and therefore the model is not robust enough to the noise. It can be seen from the contours in the last two plots: Contours in the plot above are more regular than these in the last plot.

In short, SVM with kernel performs preferably as a non-linear classifier.