EE546 HW1

1. Proof: From gradient descent update $x_{t+1} = x_t - \eta \nabla f(x_t)$

we know that $\|x_{t+1} - x^*\|_{l_2}^2 = \|x_t - \eta \nabla f(x_t) - x^*\|_{l_2}^2 = \|x_t - x^*\|_{l_2}^2 - 2\eta \langle \nabla f(x_t), x_t - x^* \rangle + \eta^2 \|\nabla f(x_t)\|_{l_2}^2$

$\therefore \|x_t - x^*\|_{l_2}^2 = \|x_{t+1} - x^*\|_{l_2}^2 + 2\eta \langle \nabla f(x_t), x_t - x^* \rangle - \eta^2 \|\nabla f(x_t)\|_{l_2}^2$

Let $t=0$, we have

$$\|x_0 - x^*\|_{l_2}^2 = \|x_1 - x^*\|_{l_2}^2 + 2\eta \langle \nabla f(x_0), x_0 - x^* \rangle - \eta^2 \|\nabla f(x_0)\|_{l_2}^2 \cdots (1)$$

since we know that $\langle \nabla f(x), x - x^* \rangle \geq \frac{1}{\beta} \|x - x^*\|_{l_2}^2 + \frac{1}{\beta} \|\nabla f(x)\|_{l_2}^2 \cdots (2)$

due to $\beta > 0$, $0 < \eta < \frac{2}{\beta} \Rightarrow \frac{1}{\beta} > \frac{\eta}{2} > 0$

(2) becomes $\langle \nabla f(x), x - x^* \rangle \geq \frac{1}{\beta} \|x - x^*\|_{l_2}^2 + \frac{\eta}{2} \|\nabla f(x)\|_{l_2}^2 \cdots (3)$

Substitute (3) into (1), we get $\|x_0 - x^*\|_{l_2}^2 \geq \|x_1 - x^*\|_{l_2}^2 + 2\eta \left[\frac{1}{\beta} \|x_0 - x^*\|_{l_2}^2 + \frac{\eta}{2} \|\nabla f(x_0)\|_{l_2}^2 \right] - \eta^2 \|\nabla f(x_0)\|_{l_2}^2$

$$\therefore \|x_0 - x^*\|_{l_2}^2 \geq \|x_1 - x^*\|_{l_2}^2 + \frac{2\eta}{\beta} \|x_0 - x^*\|_{l_2}^2$$

Then we have $\|x_1 - x^*\|_{l_2}^2 \leq (1 - \frac{2\eta}{\beta}) \|x_0 - x^*\|_{l_2}^2 \cdots (4)$

$\|x_2 - x^*\|_{l_2}^2 \leq (1 - \frac{2\eta}{\beta}) \|x_1 - x^*\|_{l_2}^2 \cdots (5)$

$\vdots$

$\|x_t - x^*\|_{l_2}^2 \leq (1 - \frac{2\eta}{\beta}) \|x_{t-1} - x^*\|_{l_2}^2 \cdots (t+3)$

Multiply equations (4), (5) $\cdots$ $(t+3)$:

we can finally get

$$\|x_t - x^*\|_{l_2}^2 \leq (1 - \frac{2\eta}{\beta})^t \|x_0 - x^*\|_{l_2}^2$$

When $x_0 = x^*$, $\|x_t - x^*\|_{l_2}^2 = (1 - \frac{2\eta}{\beta})^t \|x_0 - x^*\|_{l_2}^2 = 0$

Therefore, $\|x_t - x^*\|_{l_2}^2 \leq (1 - \frac{2\eta}{\beta})^t \|x_0 - x^*\|_{l_2}^2$

2. Proof:

(a) Since $f(x - u\nabla f(x)) \le f(x) - u(1 - \frac{uL}{2})\|\nabla f(x)\|_2^2$

Let $x = x_t$,

we have $f(x_t - u\nabla f(x_t)) \le f(x_t) - u(1 - \frac{uL}{2})\|\nabla f(x_t)\|_2^2 \cdots (1)$

Due to $x_{t+1} = x_t - u\nabla f(x_t)$

so (1) becomes $f(x_{t+1}) \le f(x_t) - u(1 - \frac{uL}{2})\|\nabla f(x_t)\|_2^2$

$f(x_t) - f(x_{t+1}) \ge u(1 - \frac{uL}{2})\|\nabla f(x_t)\|_2^2$

since $u \le \frac{1}{L}$, $u(1 - \frac{uL}{2}) \le \frac{1}{L}(1 - \frac{1}{2}) = \frac{1}{2L}$

Therefore $f(x_t) - f(x_{t+1}) \ge \frac{1}{2L}\|\nabla f(x_t)\|_2^2 \cdots (2)$

$f(x_{t-1}) - f(x_t) \ge \frac{1}{2L}\|\nabla f(x_{t-1})\|_2^2$

$\vdots$

$f(x_1) - f(x_2) \ge \frac{1}{2L}\|\nabla f(x_1)\|_2^2$

$\Rightarrow f(x_1) - f(x_{t+1}) \ge \frac{1}{2L}\sum_{i=1}^{t}\|\nabla f(x_i)\|_2^2 \ge \frac{t}{2L}\|\nabla f(x_t)\|_2^2$

using the fact that $\|\nabla f(x_i)\|_2^2$ keeps decreasing at every iteration

we get $\frac{2L}{t}[f(x_1) - f(x_{t+1})] \ge \|\nabla f(x_t)\|_2^2$

$\lim_{t\to\infty} \frac{2L}{t}[f(x_1) - f(x_{t+1})] = 0$

so $\lim_{t\to\infty}\|\nabla f(x_t)\|_2^2 = 0$

(b). Not necessarily converge to a fixed point.

Suppose $\nabla f(x_t) = \frac{1}{t}$, then $\lim_{t\to\infty}\|\nabla f(x_t)\|_2 = 0$

but $x_t = x_0 - \frac{1}{u}\sum_{m=1}^{t}\frac{1}{m}$

x_t will fail to converge to a fixed point.

(c) We have $\|\nabla f(x_t)\|_2^2 \ge r(f(x_t) - f(x^*))$

From (2): $2L[f(x_t) - f(x_{t+1})] \ge \|\nabla f(x_t)\|_2^2$

so $2L[f(x_t) - f(x_{t+1})] \ge r(f(x_t) - f(x^*))$

due to $u \le \frac{1}{L} \to L \le \frac{1}{u}$

$\therefore \frac{2}{u}[f(x_t) - f(x_{t+1})] \ge r(f(x_t) - f(x^*))$

$f(x_t) - f(x_{t+1}) \ge \frac{ur}{2}f(x_t) - \frac{ur}{2}f(x^*)$

$f(x_{t+1}) - f(x^*) \le f(x_t) - \frac{ur}{2}f(x_t) + \frac{ur}{2}f(x^*) - f(x^*)$

Hence $f(x_{t+1}) - f(x^*) \le (1 - \frac{ur}{2})(f(x_t) - f(x^*))$

Problem3: Logistic regression with momentum

(a) First read in the data from the file "wdbc.data", remove the patient i.d. and separate the features from the outputs. I use Python to implement this step as shown below:

```
def input_data(datafile_w, data_m, data_n):
    """
    Inputs:
        the location of the data file
        the number of row
        the number of column
    Outputs:
        feature matrix X
        label vector y
    """
    InputData = np.zeros((data_m, data_n), dtype='float')
    with open(datafile_w, 'r') as f:
        reader = csv.reader(f)
        for i, row in enumerate(reader):
            data = [float(datum) for datum in row]
            InputData[i] = data
    print("InputData.shape", InputData.shape)

    Data_X = InputData[0:data_m, 2:32]  # Feature matrix X
    Data_y = InputData[0:data_m, 1]  # Label vector y
    Data_y = Data_y.astype(np.int)
    Data_y = np.expand_dims(Data_y, axis=1)
    print("Data_X.shape", Data_X.shape)
    print("Data_y.shape", Data_y.shape)

    return Data_X, Data_y


from hw1_utils import input_data
Data_X, Data_y = input_data(datafile_w='wdbc.data', data_m=569, data_n=32)
```

After running the code, I can know that the shape of the feature matrix "Data_X" is (569, 30) and the shape of the label vector "Data_y" is (569, 1).

(b) In this part, I will normalize the data. First of all, calculate the mean vector of patient features across all of the data set, which is

$$\bar{x} = \frac{1}{569}\sum_{i=1}^{569} x_i.$$

Next, subtract the mean from each of the features, which is $x_i \leftarrow x_i - \bar{x}$. Finally, normalize the data via $x_i \leftarrow \frac{x_i}{\|x_i\|_{l_2}}$. The Python code is

```
def normalize_data(Data_X, sample_num):
    """
    Inputs:
        the feature matrix X
        the number of samples
    Outputs:
        normalized feature matrix X
    """
```

```
    X_mean = np.mean(Data_X, axis=0)
    X_mean = np.expand_dims(X_mean, axis=0)
    Data_mean = np.repeat(a=X_mean, repeats=sample_num, axis=0)

    Data_X = Data_X - Data_mean
    Data_L2 = np.expand_dims(np.sqrt(np.sum(np.square(Data_X), axis=1)), axis=1)
    Data_X = np.true_divide(Data_X, Data_L2)

    return Data_X


from hw1_utils import normalize_data
Data_x = normalize_data(Data_X, sample_num=569)
```

After running the code, I will normalize the data set.

(c) In this part, I am about to partition the normalized data into train/test sets at random 100 times. In each trial by setting $\lambda = 0.01$, I will run gradient descent for T = 500 iterations and then use the trained model to make predictions on the test data and calculate the average error (average number of miss-classified patients on the test data) for each trial. Report the average over the 100 trials.

Since I have

$$f(w,b) := \sum_{i=1}^{N}(-y_i(w^T x_i + b) + \ln(1+\exp(w^T x_i + b))) + \frac{\lambda}{2}\|w\|_{l_2}^2,$$

Then I can get

$$\nabla f(w) = \sum_{i=1}^{N}\left(-y_i x_i + \frac{x_i \exp(w^T x_i + b)}{1+\exp(w^T x_i + b)}\right) + \lambda w \text{ and}$$

$$\nabla f(b) = \sum_{i=1}^{N}\left(-y_i + \frac{\exp(w^T x_i + b)}{1+\exp(w^T x_i + b)}\right).$$

Therefore

$$w_{t+1} = w_t - \mu\left(\sum_{i=1}^{N}\left(-y_i x_i + \frac{x_i \exp(w_t^T x_i + b)}{1+\exp(w_t^T x_i + b)}\right) + \lambda w_t\right) \text{ and}$$

$$b_{t+1} = b_t - \mu\sum_{i=1}^{N}\left(-y_i + \frac{\exp(w^T x_i + b_t)}{1+\exp(w^T x_i + b_t)}\right).$$

The Python code of partitioning the data set is

```
def split_tr_te(dataset, label_set, train_num, seed_value):
    """
    Inputs:
        the total dataset
        the total label set
        the number of the training set
        the seed value to random sampling
    Output: the training set x & y and the test set x & y
    """
    sample_num, fea_num = dataset.shape[:]
    index_list = list(range(sample_num))
    random.seed(seed_value)  # Make the shuffle function behave the same
    random.shuffle(index_list)
    train_index = index_list[0: train_num]
    test_index = index_list[train_num: sample_num]

    train_x = np.zeros((train_num, fea_num), dtype='float')
    train_y = np.zeros((train_num, 1), dtype='int')
    for sample_i in range(train_num):
        train_x[sample_i] = dataset[train_index[sample_i]]
```

```
        train_y[sample_i] = label_set[train_index[sample_i]]

    test_num = sample_num - train_num
    test_x = np.zeros((test_num, fea_num), dtype='float')
    test_y = np.zeros((test_num, 1), dtype='int')
    for sample_i in range(test_num):
        test_x[sample_i] = dataset[test_index[sample_i]]
        test_y[sample_i] = label_set[test_index[sample_i]]

    return train_x, train_y, test_x, test_y


from hw1_utils import split_tr_te
train_x = np.zeros((100, 500, Data_x.shape[1]), dtype='float')
train_y = np.zeros((100, 500, Data_y.shape[1]), dtype='int')
test_x = np.zeros((100, 69, Data_x.shape[1]), dtype='float')
test_y = np.zeros((100, 69, Data_y.shape[1]), dtype='int')
for index_i in range(100):
    train_x[index_i], train_y[index_i], test_x[index_i], test_y[index_i] = \
        split_tr_te(dataset=Data_x, label_set=Data_y, train_num=500,
seed_value=index_i)
print("train_x.shape", train_x.shape)
print("train_y.shape", train_y.shape)
print("test_x.shape", test_x.shape)
print("test_y.shape", test_y.shape)
```

After running the code, I can get the training set x, y as (100, 500, 30), (100, 500, 1) and test set as (100, 69, 30), (100, 69, 1). The Python code to run gradient descent is

```
def fit_transform(self, w, b, iter_num, test_x, test_y):
    """
        Train the model and test it
    :param w: The w coefficient
    :param b: The b coefficient
    :param iter_num: The number of iterations
    :param test_x: The feature matrix of the test set
    :param test_y: The label vector of the test set
    :return: The accuracy of testing patients
    """
    accuracy = np.zeros((self.train_x.shape[0], 1))  # 100 trials
    for trial_i in range(self.train_x.shape[0]):
        for iter_i in range(iter_num):
            total_w = np.zeros((30, 1))
            total_b = 0
            for train_i in range(self.train_x.shape[1]):
                inner_part = dot(w.T, self.train_x[trial_i, train_i].T) + b
                exp_part = exp(inner_part)
                total_w = total_w - \
                          self.train_y[trial_i, train_i] * \
                          expand_dims(self.train_x[trial_i, train_i].T, axis=1) +
\
                          expand_dims(self.train_x[trial_i, train_i].T, axis=1) *
\
                          exp_part / (1 + exp_part)
```

```
                total_b = total_b - self.train_y[trial_i, train_i] + exp_part /
(1 + exp_part)

            total_w = total_w + self.lamb_da * w
            w = w - self.lr * total_w
            b = b - self.lr * total_b

        accuracy_i = np.zeros(test_y.shape[:], dtype='int')
        for test_i in range(test_x.shape[1]):
            test_p = 1 / (1 + exp(- np.dot(w.T, test_x[trial_i, test_i].T) - b))
            if test_p >= 0.5:
                predict_label = 1
            else:
                predict_label = 0
            if predict_label == test_y[trial_i, test_i]:
                accuracy_i[0, test_i] = 1
        accuracy[trial_i] = accuracy_i[0].mean()
    print("The accuracy of testing patients is %f" % (accuracy.mean()))

b0 = 0.05

np.random.seed(1)
w0 = np.random.normal(0, 0.5, size=(30, 1))

from hw1_utils import logi_regre
Hmwk1 = logi_regre(train_x=train_x, train_y=train_y, lr=0.01, lamb_da=0.01)
Hmwk1.fit_transform(w=w0, b=b0, iter_num=500, test_x=test_x, test_y=test_y)
```

After running the code, I can know over the 100 trials the average error number is 4.

(d) In this part, I am about to perform the experiment with the same step size as before but now report the number of iterations it takes to get to an accuracy of 10^{-6} calculated via the first iteration t when the following inequality holds

$$\|\nabla f(w_t, b_t)\|_{l_2}^2 \leq 10^{-6}(1 + |f(w_t, b_t)|),$$

where

$$\|\nabla f(w_t, b_t)\|_{l_2}^2 = \|\nabla f(w_t)\|_{l_2}^2 + \|\nabla f(b_t)\|_{l_2}^2 \text{ and}$$

$$|f(w_t, b_t)| = \sum_{i=1}^{N} |(-y_i(w_t{}^T x_i + b_t) + \ln(1 + \exp(w_t{}^T x_i + b_t)))| + \frac{\lambda}{2}\|w_t\|_{l_2}^2.$$

The Python code to check iterations of gradient descent and two other algorithms is

```
def check_iterations(self, w, b, epsilon, eta, method, verbose=False):
    """
        Check how many iterations to converge
            :param w: The w coefficient
    :param b: The b coefficient
    :param epsilon: The value of epsilon
    :param eta: The value of eta1
    :param method: 'gradent descent', 'heavy ball', or 'Nesterov'
    :param verbose: "True" or "False" to print the situation
    :return: The average iterations for three different algorithms
    """
    iter_num = np.zeros((self.train_x.shape[0], 1))  # (100, 0)
    for trial_i in range(self.train_x.shape[0]):
        iter_i = 0
```

```
        while True:
            total_w = np.zeros((30, 1))
            total_b = 0
            total_wb = 0
            for train_i in range(self.train_x.shape[1]):
                inner_part = dot(w.T, self.train_x[trial_i, train_i].T) + b
                exp_part = exp(inner_part)

                total_w = total_w - \
                          self.train_y[trial_i, train_i] * \
                          expand_dims(self.train_x[trial_i, train_i].T, axis=1) +
\
                          expand_dims(self.train_x[trial_i, train_i].T, axis=1) *
\
                          exp_part / (1 + exp_part)
                total_b = total_b - self.train_y[trial_i, train_i] + exp_part /
(1 + exp_part)
                total_wb = total_wb + absolute(- self.train_y[trial_i, train_i] *
inner_part +
                                               log(1 + exp_part))

            total_w = total_w + self.lamb_da * w
            total_wb = total_wb + 0.5 * self.lamb_da * sum(square(w))
            if method == 'gradient descent':
                w = w - self.lr * total_w
                b = b - self.lr * total_b

            elif method == 'heavy ball':
                if iter_i == 0:
                    w_1 = w
                    w = w - self.lr * total_w
                    b_1 = b
                    b = b - self.lr * total_b
                else:
                    temp_w = w - w_1
                    w_1 = w
                    w = w - self.lr * total_w + eta * temp_w

                    temp_b = b - b_1
                    b_1 = b
                    b = b - self.lr * total_b + eta * temp_b

            elif method == 'Nesterov':
                if iter_i == 0:
                    temp_w_y = - self.lr * total_w
                    temp_w = w + temp_w_y
                    w = temp_w + eta * temp_w_y

                    temp_b_y = - self.lr * total_b
                    temp_b = b + temp_b_y
                    b = temp_b + eta * temp_b_y
                else:
```

```
                    temp_w_y = - self.lr * total_w + eta * temp_w_y
                    temp_w = w + temp_w_y
                    w = temp_w + eta * temp_w_y

                    temp_b_y = - self.lr * total_b + eta * temp_b_y
                    temp_b = b + temp_b_y
                    b = temp_b + eta * temp_b_y

            else:
                print("Please input the correct method!")
                return False

            iter_i += 1

            upper_part = sum(square(total_w)) + sum(square(total_b))
            lower_part = 1 + total_wb
            if upper_part / lower_part <= epsilon:
                iter_num[trial_i] = iter_i
                if verbose:
                    print("The %d/%d is finished!" % (trial_i+1,
self.train_x.shape[0]))
                break

    print("The average number of converging iterations is %f for the %s method."
          % (iter_num.mean(), method))


Hmwk1.check_iterations(w=w0, b=b0, epsilon=1e-6, eta=0.8, method='gradient
descent')
```

After running the code, I can know over the 100 trials the average iteration number is 10039. I just run the gradient descent of w and b simultaneously for every iteration to get this iteration number which is much fewer than the standard number of the professor's.

(e) In this part, I am about to perform the same experiment but now add a momentum term (1) using the heavy ball method and (2) using Nesterov's accelerated scheme. In both cases keep the same step size as before but fine tune the momentum parameter to get the smallest number of iterations for convergence based on the stopping criteria above. The sample Python code is

```
Hmwk1.check_iterations(w=w0, b=b0, epsilon=1e-6, eta=0.6, method='heavy ball')
Hmwk1.check_iterations(w=w0, b=b0, epsilon=1e-6, eta=0.5, method='Nesterov')
```

Table 1: the number of iterations for heavy ball and Nesterov for different momentum values

eta	0.8	0.85	0.89	0.9	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99
heavy ball	2006	1502	1099	998	896	795	693	591	489	391	336	388	736
Nesterov	1113	811	579	522	466	410	355	302	257	241	269	377	743

After running the code, form the Table 1, I can know over the 100 trials the smallest average iteration number is 336 for heavy ball and 241 for Nesterov. The convergence curves of three algorithms is shown in the Figure 1. The convergence curves for gradient descent, heavy ball and Nesterov are shown in the Figure 2, 3, and 4 individually. From the Table 1 we know that Nesterov method does converge faster. Moreover, from the Figure 1 we can see that Nesterov has more capability of jumping out of the local optimum when dealing with non-convex problems. Therefore, I prefer the Nesterov method.

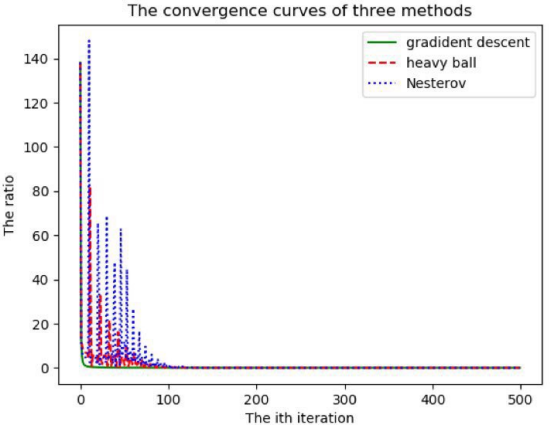

Figure 1: The converge curves of three methods

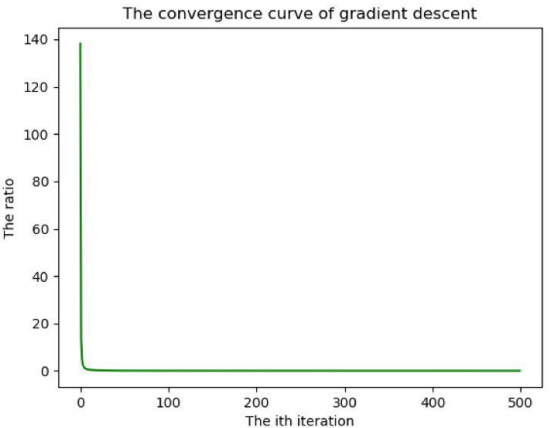

Figure 2: The converge curve of gradient descent

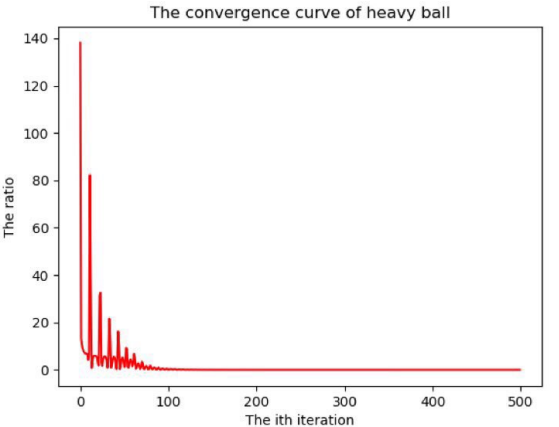

Figure 3: The converge curve of heavy ball

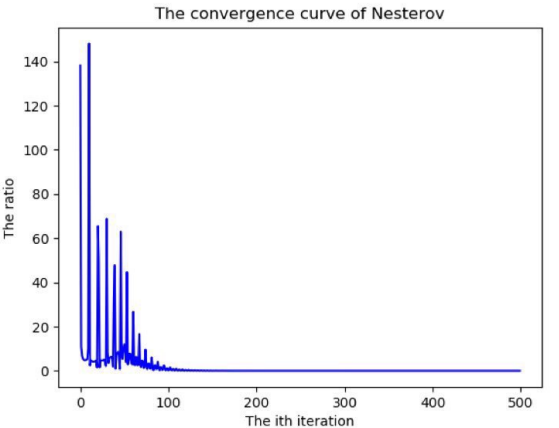

Figure 4: The converge curve of heavy ball