EE475 HW3

Name: Zhisheng Lin
student #: 3226299

Exercise 4.3.

(a) $g_2(\tilde{w}) = \sum_{p=1}^{P} \log\left(1+e^{-y_p \tilde{x}_p^T \tilde{w}}\right) \qquad \tilde{x}_p = \begin{bmatrix} 1 \\ x_p \end{bmatrix} \quad \tilde{w} = \begin{bmatrix} b \\ w \end{bmatrix}$

$$\frac{\partial}{\partial w_j} g_2(\tilde{w}) = \sum_{p=1}^{P} \frac{\partial}{\partial w_j} \log\left(1+e^{-y_p \tilde{x}_p^T \tilde{w}}\right) = -\sum_{p=1}^{P} \frac{1}{1+e^{-y_p \tilde{x}_p^T \tilde{w}}} \cdot e^{-y_p \tilde{x}_p^T \tilde{w}} \cdot \frac{\partial}{\partial w_j}\left(y_p \tilde{x}_p^T \tilde{w}\right)$$

$$= -\sum_{p=1}^{P} \frac{1}{\left(1+e^{-y_p \tilde{x}_p^T \tilde{w}}\right) \cdot e^{y_p \tilde{x}_p^T \tilde{w}}} \cdot y_p \tilde{x}_p = -\sum_{p=1}^{P} \frac{1}{e^{y_p \tilde{x}_p^T \tilde{w}}+1} \cdot y_j x_j$$

$\therefore \nabla g_2(\tilde{w}) = -\sum_{p=1}^{P} \sigma\left(-y_p \tilde{x}_p^T \tilde{w}\right) y_p \tilde{x}_p$

(b) $r_p = -\sigma\left(-y_p \tilde{x}_p^T \tilde{w}\right) y_p$

$\therefore r = [r_1 \ r_2 \ r_3 \cdots r_P]^T$

$\tilde{X} = [\tilde{x}_1 \ \tilde{x}_2 \cdots \tilde{x}_P]$

$\therefore \tilde{X} \cdot r = r_1 \tilde{x}_1 + r_2 \tilde{x}_2 + \cdots + r_P \tilde{x}_P = \sum_{p=1}^{P} r_p \tilde{x}_p = -\sum_{p=1}^{P} \sigma\left(-y_p \tilde{x}_p^T \tilde{w}\right) y_p \tilde{x}_p = \nabla g_2(\tilde{w})$

$\therefore \nabla g(\tilde{w}) = \tilde{X} r.$

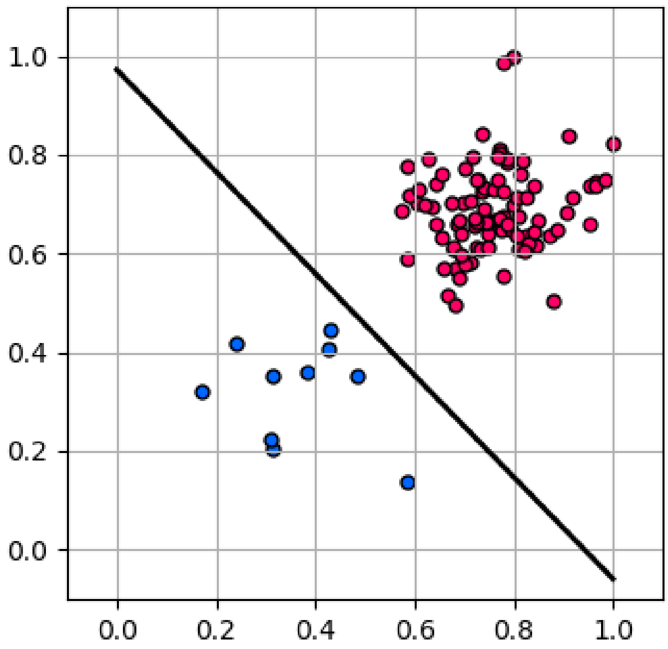
Exercise 4.3 (c)

Code:

```
# This file is associated with the book
# "Machine Learning Refined", Cambridge University Press, 2016.
# by Jeremy Watt, Reza Borhani, and Aggelos Katsaggelos.

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

# sigmoid for softmax/logistic regression minimization
def sigmoid(z):
    y = 1/(1+np.exp(-z))
    return y

# import training data
def load_data(csvname):
    # load in data
    reader = csv.reader(open(csvname, "r"), delimiter=",")
    d = list(reader)

    # import data and reshape appropriately
    data = np.array(d).astype("float")
    X = data[:,0:2]
    y = data[:,2]
    y.shape = (len(y),1)

    # pad data with ones for more compact gradient computation
    o = np.ones((np.shape(X)[0],1))
    X = np.concatenate((o,X),axis = 1)
    X = X.T

    return X,y

# YOUR CODE GOES HERE - create a gradient descent function for softmax
cost/logistic regression
def softmax_grad(X,y):
    w = np.random.randn(3,1)
    alpha = 10**(-2)
    iter = 1
    max_its = 3000
    grad = 1
    while np.linalg.norm(grad) > 10**(-12) and iter < max_its:
        e = np.dot(X.T,w)
        q = -y*e
        r_1 = -sigmoid(q)
        r_2 = r_1*y
        r = r_2
        grad = np.dot(X,r)
        w = w - alpha*grad
        iter += 1

    return w

# plots everything
def plot_all(X,y,w):
    # custom colors for plotting points
```

```
    red = [1,0,0.4]
    blue = [0,0.4,1]

    # scatter plot points
    fig = plt.figure(figsize = (4,4))
    ind = np.argwhere(y==1)
    ind = [s[0] for s in ind]
    plt.scatter(X[1,ind],X[2,ind],color = red,edgecolor = 'k',s = 25)
    ind = np.argwhere(y==-1)
    ind = [s[0] for s in ind]
    plt.scatter(X[1,ind],X[2,ind],color = blue,edgecolor = 'k',s = 25)
    plt.grid('off')

    # plot separator
    s = np.linspace(0,1,100)
    plt.plot(s,(-w[0]-w[1]*s)/w[2],color = 'k',linewidth = 2)

    # clean up plot
    plt.xlim([-0.1,1.1])
    plt.ylim([-0.1,1.1])
    plt.show()

# load in data
X,y = load_data('imbalanced_2class.csv')

# run gradient descent
w = softmax_grad(X,y)

# plot points and separator
plot_all(X,y,w)
```

Figure:



Exercise 4.5

(a) ① without multiplying C, the equation of separating hyperplane is:

$$b + x_p^T W = 0$$

multiplying C, the equation is:

$$c \cdot b + c \cdot x_p^T \cdot W = 0 \Rightarrow b + x_p^T W = 0$$

So, no change to the equation of separating hyperplane

② $g(c \cdot b, c \cdot W) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p (c \cdot b + c \cdot x_p^T W)}\right)$

Since $C > 1$, $e^{-y_p (c \cdot b + c \cdot x_p^T W)}$ become smaller.

$g(cb, cw) < g(b, w)$

(b)

① • $\underset{b, W}{\text{minimize}} \sum_{p=1}^{P} \log\left(1 + e^{-y_p (b + x_p^T W)}\right) = \underset{b, W}{\text{minimize}} \sum_{p=1}^{P} \log\left(1 + \frac{1}{e^{y_p (b + x_p^T W)}}\right)$

Since b and W is in denominator, if b and W become larger, the softmax cost will become lesser. That's why it is possible for the parameters to grow infinitely large.

• Besides, since multiplying a constant with b and with W will not change the hyperplane, so b and W can grow infinitely large.

② if b and W can become infinitely large, the code will not stop, which cannot lead to the final hyperplane.

Exercise 4.9

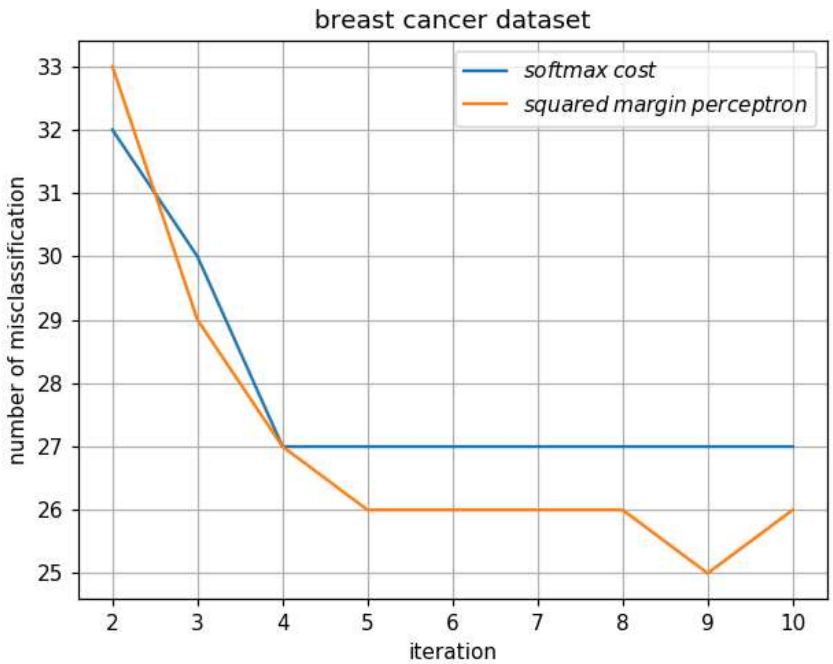
Code:

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

# sigmoid for softmax/logistic regression minimization
def sigmoid(z):
    y = 1/(1+np.exp(-z))
    return y

# import training data
def load_data(csvname):
    # load in data
    reader = csv.reader(open(csvname, "r"), delimiter=",")
    d = list(reader)

    # import data and reshape appropriately
    data = np.array(d).astype("float")
    X = data[:,0:8]
    y = data[:,8]
    y.shape = (len(y),1)

    # pad data with ones for more compact gradient computation
    o = np.ones((np.shape(X)[0],1))
    X = np.concatenate((o,X),axis = 1)
    X = X.T

    return X,y

# create a newton's method function for softmax and squared margin
def softmax_squared_newton(X,y):

    # define initial w for softmax and squared margin
    w_soft = (np.random.randn(9,1))/35
    w_squared = w_soft

    # define some common parameters
    X = X/35
    y = y
    max_its = 10

    # define parameters of softmax
    grad_soft = 1
    ite_soft = 0
    it_soft = []
    misc_soft = []

    # define parameters of squared margin
    grad_squared = 1
    ite_squared = 0
    it_squared = []
    misc_squared = []

    # softmax iteration
    while np.linalg.norm(grad_soft) > 10**(-12) and ite_soft < max_its:
        # calculate gradient
```

```
        r_0 = np.dot(X.T,w_soft)
        r_1 = -y*r_0
        r_2 = -sigmoid(r_1)
        r_3 = r_2*y
        r = r_3
        grad_soft = np.dot(X,r)

        # calculate misclassfication
        t_0 = np.dot(X.T,w_soft)
        t_1 = -y*t_0
        misclass_soft = np.sign(t_1)
        misclass_soft_new = (misclass_soft > 0).sum()
        misc_soft.append(misclass_soft_new)

        # calculate hessian
        t_2 = sigmoid(t_1)
        t_3 = 1 - t_2
        t_4 = t_2*t_3
        t_5 = t_4*(X.T)
        grad2_soft = np.dot(t_5.T,X.T)

        # calculate new w
        w_soft = w_soft - np.linalg.pinv(grad2_soft).dot(grad_soft)

        # calculate iteration
        ite_soft += 1
        it_soft.append(ite_soft)

    # squared margin iteration
    while np.linalg.norm(grad_squared) > 10**(-12) and ite_squared < 
max_its:
        # calculate gradient
        e_0 = np.dot(X.T,w_squared)
        e_1 = -y*e_0
        e_2 = (1+e_1)>0
        e_3 = e_2*(1+e_1)
        e = e_2*y
        grad_squared = -2*np.dot(X,e)

        # calculate misclassification
        f_0 = np.dot(X.T,w_squared)
        f_1 = -y*f_0
        misclass_squared = np.sign(e_1)
        misclass_squared_new = (misclass_squared > 0).sum()
        misc_squared.append(misclass_squared_new)

        # calculate hessian
        omega = e_1 > -1
        X_new = (X.T)*omega
        X_new = X_new.T
        grad2_squared = 2 * np.dot(X_new,X_new.T)

        # calculate new w
        w_squared = w_squared - 
np.linalg.pinv(grad2_squared).dot(grad_squared)

        # calculate iteration
        ite_squared += 1
        it_squared.append(ite_squared)
```

```python
    return it_soft,misc_soft,it_squared,misc_squared


# plots everything
def plot_all(a_ite,a_misclass,b_ite,b_misclass):

    plt.figure(dpi=110,facecolor='w')
    plt.plot(a_ite[1:],a_misclass[1:])
    plt.plot(b_ite[1:],b_misclass[1:])
    plt.title('breast cancer dataset')
    plt.xlabel('iteration')
    plt.ylabel('number of misclassification')
    plt.legend([r'$softmax\:cost$',r'$squared\:margin\:perceptron$'])
    plt.grid(True)
    plt.show()

# load in data
X,y = load_data('breast_cancer_data.csv')

# get the iteration and misclassification of softmax
a = softmax_squared_newton(X,y)
a_ite = a[0]
a_misclass = a[1]

# get the iteration and misclassification of squared margin
b = softmax_squared_newton(X,y)
b_ite = b[2]
b_misclass = b[3]


# plot points and separator
plot_all(a_ite,a_misclass,b_ite,b_misclass)
```

Figure:



Exercise 4.12

① when $y_p = +1$ $\bar{y}_p = 1$

Suppose $y_p = +1$, then $g(b, w) = \sum_{p=1}^{P} \log(1 + e^{-(b + x_p^T w)})$

Substituting $\bar{y}_p = 1$, then $h(b, w) = -\sum_{p=1}^{P} \log \sigma(b + x_p^T w)$

$$= \sum_{p=1}^{P} \log\left(\frac{1}{1 + e^{-(b + x_p^T w)}}\right)^{-1}$$

$$= \sum_{p=1}^{P} \log(1 + e^{-(b + x_p^T w)})$$

So, $g(b, w) = h(b, w)$

② when $y_p = -1$, $\bar{y}_p = 0$

$g(b, w) = \sum_{p=1}^{P} \log(1 + e^{(b + x_p^T w)})$

$h(b, w) = -\sum_{p=1}^{P} \log(1 - \sigma(b + x_p^T w))$

$$= -\sum_{p=1}^{P} \log \frac{e^{-(b + x_p^T w)}}{1 + e^{-(b + x_p^T w)}}$$

$$= \sum_{p=1}^{P} \log(1 + e^{-(b + x_p^T w)}) + (b + x_p^T w)$$

in this case, $g(b, w) \neq h(b, w)$

Exercise 4.16

※Method 1: OvA

Code:

```
from __future__ import print_function
import numpy as np
import csv

def sigmoid(t):
    return 1/(1 + np.exp(-t))

def checkSize(w, X, y):
        # w and y are column vector, shape [N, 1] not [N,]
        # X is a matrix where rows are data sample
        assert X.shape[0] == y.shape[0]
        assert X.shape[1] == w.shape[0]
        assert len(y.shape) == 2
        assert len(w.shape) == 2
        assert w.shape[1] == 1
        assert y.shape[1] == 1

def compactNotation(X):
        return np.hstack([np.ones([X.shape[0], 1]), X])

def readData(path):
        """
        Read data from path (either path to MNIST train or test)
        return X in compact notation (has one appended)
        return Y in with shape [10000,1] and starts from 0 instead of 1
        """
        # Read data from path (either path to MNIST train or test)
        reader = csv.reader(open(path, "r"),delimiter=",")
        d = list(reader)
        # return X in compact notation (has one appended)
        # return Y in with shape [10000,1] and starts from 0 instead of
1
        data = np.array(d).astype("float")
        X = data[:,:-1]
        Y = data[:,-1]
        Y.shape = (len(Y),1)
        X = compactNotation(X)

        return X,Y

def softmaxGrad(w, X, y):
        checkSize(w, X, y)
        ### RETURN GRADIENT
        X = X.T
        a_0 = np.dot(X.T,w)
        a_1 = -y*a_0
        a_2 = -sigmoid(a_1)
        a_3 = a_2*y
        grad = np.dot(X,a_3)

        return grad

def accuracy(OVA, X, y):
```

```
        """
        Calculate accuracy using matrix operations!
        """
        X = X.T
        ylab = np.array([np.argmax(np.dot(X.T,OVA),axis=1)]).T
        I = (y==ylab)
        accu = 1 - (1/len(y))*(I.sum())

        return accu

def gradientDescent(grad, w0, *args, **kwargs):
        max_iter = 5000
        alpha = 0.001
        eps = 10^(-5)

        w = w0
        iter = 0
        while True:
                gradient = grad(w, *args, **kwargs)
                w = w - alpha * gradient

                if iter > max_iter or np.linalg.norm(gradient) < eps:
                        break

                if iter  % 1000 == 1:
                        print("Iter %d " % iter)

                iter += 1

        return w

def oneVersusAll(Y, value):
        """
        generate label Yout,
        where Y == value then Yout would be 1
        otherwise Yout would be -1
        """
        a = Y==value
        Yout_1 = Y*a
        b = Y!=value
        Yout_2 = Y*b
        Yout = Yout_1+Yout_2

        return Yout

if __name__=="__main__":

        trainX, trainY = readData('MNIST_data/MNIST_train_data.csv')

        # training individual classifier
        Nfeature = trainX.shape[1]
        Nclass = 10
        OVA = np.zeros((Nfeature, Nclass))
        for i in range(Nclass):
                print("Training for class " + str(i))
                w0 = np.random.rand(Nfeature, 1)
                OVA[:, i:i+1] = gradientDescent(softmaxGrad, w0, trainX,
oneVersusAll(trainY, i))
```

```
        print("Accuracy for training set is: %f" % accuracy(OVA, trainX,
trainY))

        testX, testY = readData('MNIST_data/MNIST_test_data.csv')
        print("Accuracy for test set is: %f" % accuracy(OVA, testX,
testY))
```

Result:

Accuracy for training set is: 0.924967

Accuracy for test set is: 0.919300

※Method 2: Multiclass softmax

Code of multiClassSoftmax:

```
import numpy as np

def checkSize(w, X, y):
        # w: 785 by 10 matrix
        # X: N by 785 matrix
        # y: N by 1 matrix
        assert y.dtype == 'int'
        assert X.shape[0] == y.shape[0]
        assert X.shape[1] == w.shape[0]
        assert len(y.shape) == 2
        assert y.shape[1] == 1

def loss(w, X, y):
        """
        Optional
        Useful to run gradient checking
        Utilize softmax function below
        """
        checkSize(w, X, y)

def grad(w, X, y):
        """
        Return gradient of multiclass softmax
        Utilize softmax function below
        """
        checkSize(w, X, y)
        a = np.array([np.argmax(np.dot(X,w),axis=1)]).T
        b = np.dot(X,w)
        i = 0
        for i in len(y)
                b[i,a[i,0]] = b[i,a[i,0]]-1
                i += 1
        grad = np.dot(X.T,(softmax(w,X)-b))

        return grad


def softmax(w, X):
        scores = np.matmul(X, w)
```

```
        maxscores = scores.max(axis = 1)
        scores = scores - maxscores[:, np.newaxis]
        exp_scores = np.exp(scores)

        sum_scores = np.sum(exp_scores, axis = 1)
        return exp_scores/sum_scores[:, np.newaxis]

def predict(w, X):
        """
        Prediction
        """
        X = X.T
        ylab = np.array([np.argmax(np.dot(X.T,w),axis=1)]).T
        I = (y==ylab)

        return I

def accuracy(w, X, y):
        """
        Accuracy of the model
        """
        accu = 1 - (1/len(y))*(predict(w,X).sum())

        return accu
```

Code of train:

```
import numpy as np
from multiClassSoftmax import *
import csv

def compactNotation(X):
        """
        append 1 to X
        """
        return np.hstack([np.ones([X.shape[0], 1]), X])

def readData(path):
        """
        Read data from a specified path
        Returns:
                X: in compact notation
                Y: a matrix of [Nsamples, 1] where values are from 0 to
9
        """
        # Read data from path (either path to MNIST train or test)
        reader = csv.reader(open(path, "r"),delimiter=",")
        d = list(reader)
        # return X in compact notation (has one appended)
        # return Y: a matrix of [Nsamples, 1] where values are from 0 to
9
        data = np.array(d).astype("float")
        X = data[:,:-1]
        Y = data[:,-1]
        Y.shape = (len(Y),1)
        X = compactNotation(X)

        return X,Y
```

```python
def checkGradient(loss, grad, w, *args):
        """
        Gradient checking
        """
        computed_grad = grad(w, *args)

        # compute numerical gradient
        num_grad = np.zeros_like(computed_grad)
        eps = 1e-5
        for i in range(computed_grad.shape[0]):
                for j in range(computed_grad.shape[1]):
                        w1 = w.copy()
                        w1[i][j] += eps
                        num_grad[i][j] = (loss(w1, *args) - loss(w,
*args))/eps

        assert np.linalg.norm(computed_grad -
num_grad)/np.linalg.norm(computed_grad + num_grad) < 1e-2

def gradientDescent(grad, w0, *args, **kwargs):
        """
        Gradient descent
        """
        max_iter = 5000
        alpha = 0.001
        eps = 1e-5

        w = w0
        iter = 0
        while True:
                gradient = grad(w, *args,**kwargs)
                w = w - alpha * gradient

                if iter > max_iter or np.linalg.norm(gradient) < eps:
                        break

                if iter  % 1000 == 1:
                        print("Iter %d " % iter)

                iter += 1

        return w


if __name__ == "__main__":

        trainX, trainY = readData('MNIST_data/MNIST_train_data.csv')
        testX, testY = readData('MNIST_data/MNIST_test_data.csv')

        # # gradient checking
        # w = np.ones((785,10))
        # print("Checking gradient")
        # checkGradient(loss, grad, w, testX, testY)

        # Optimizing with gradient descent
        w0 = np.random.rand(785,10)
        w_optimal = gradientDescent(grad, w0, testX, testY)

        # Accuracy
```

```
        print("Accuracy for training set is %f" % accuracy(w_optimal, trainX, trainY))
        print("Accuracy for test set is %f" % accuracy(w_optimal, testX, testY))
```

Result:

Accuracy for training set is: 0.935967

Accuracy for test set is: 0.929300