CSE446 HW2

Brian Kang*†

Conceptual Questions

A.0

a. Suppose that your estimated model for predicting house prices has a large positive weight on 'number of bathrooms'. Does it implies that if we remove the feature "number of bathrooms" and refit the model, the new predictions will be strictly worse than before? Why?

- Not necessarily. There is a possibility that removing that feature would decrease complexity of the model where we already had serious overfitting. This can lead better predictions. But, given that we are predicting house prices, keeping "number of bathrooms" sounds like a good idea to me.

b. Compared to L2 norm penalty, explain why a L1 norm penalty is more likely to result in a larger number of 0s in the weight vector or not?

- The L2 norm penalizes "more smoothly" in a sense. Recall the graph with a diamond constraint region for L1 and circle for L2 and a SSR contour plot for each. For L1 norm, we keep the y-axis β_2 value while the x-axis $\beta_1 = 0$ with the penalization (leading to many 0s in the weight vector). But, for the L2 norm, β_1 is close to zero and β_2 is close to the apex of the circle.

c. In at most one sentence each, state one possible upside and one possible downside of using the following regularizer: $\sum_i |w_i|^{0.5}$

- Like the lasso, the L(1/2) norm can help with feature selection by shrinking some weights to zero; however, the downside could be pushing too many weights down to zero since the penalty constraint region is smaller than L1.

d. True or False: If the step-size for gradient descent is too large, it may not converge.

- True. By keeping the step-size too large, we may head in the direction of the optimal point, but just largely bypass it and keep repeating it.

e. In your own words, describe why SGD works.

- SGD works in the sense that we don't need all of the data points to get an idea of the direction towards the minimum point. Getting a general direction from one point each step at a time is sufficient.

*Collaborated with Cindy Wu

†References: An Introduction to Statistical Learning (James, Witten, Hastie, Tibshirani)

f. In at most one sentence each, state one possible advantage of SGD (stochastic gradient descent) over GD (gradient descent) and one possible disadvantage of SGD relative to GD.

- Benefit: Randomly choosing one point to get a general direction and repeating that many times to approach the minimum is very cost efficient compared to GD.
- Cost: SGD naturally involves more noise and so may require much more steps/updates than GD to approach the optimal value.

A.1

a. *Proof.*

i $\forall i \in \{1, \ldots, n\}$, $|x_i| \geq 0$, so $f(x) = \sum_{i=1}^{n} |x_i| \geq 0$. If $|x_i| = 0$ $\forall i \in \{1, \ldots, n\}$, then $f(x) = 0$.

ii $f(ax) = \sum_{i=1}^{n} |ax_i| = |a| \sum_{i=1}^{n} |x_i| = |a| f(x)$

iii $f(x+y) = \sum_{i=1}^{n} |x_i + y_i| \leq \sum_{i=1}^{n} |x_i| + \sum_{i=1}^{n} |y_i| = f(x) + f(y)$

Therefore, $f(x) = \sum_{i=1}^{n} |x_i|$ is a norm. □

b. *Proof.* Let $x, y \in \mathbb{R}^2$, $x, y \geq 0$. Then,

$$
\begin{aligned}
g(x+y) &= (|x_1 + y_1|^{1/2} + |x_2 + y_2|^{1/2})^2 \\
&= |x_1 + y_1| + |x_2 + y_2| + 2 * |x_1 + y_1|^{1/2} * |x_2 + y_2|^{1/2} \\
&= |x_1 + x_2| + |y_1 + y_2| + 2 * |x_1 + y_1|^{1/2} * |x_2 + y_2|^{1/2} \\
&= (|x_1 + x_2|^{1/2})^2 + (|y_1 + y_2|^{1/2})^2 + 2 * |x_1 + y_1|^{1/2} * |x_2 + y_2|^{1/2} \\
&\nleq (|x_1 + x_2|^{1/2})^2 + (|y_1 + y_2|^{1/2})^2 = g(x) + g(y)
\end{aligned}
$$

Thus, $\exists x, y \geq 0$ such that $g(x+y) \nleq g(x) + g(y)$, i.e., $g(x) = \sum_{i=1}^{n} (|x_i|^{1/2})^2$ is not a norm. □

A.2

I is not convex because $\lambda b + (1-\lambda)c \notin$ I.

II is convex.

III is not convex because $\lambda a + (1-\lambda)d \notin$ III.

A.3

a. I is convex.

b. II is not convex because $f(\lambda a + (1-\lambda)b) \nleq \lambda f(a) + (1-\lambda) f(b)$.

c. III is not convex on $[a, d]$ because $f(\lambda a + (1-\lambda)c) \nleq \lambda f(a) + (1-\lambda) f(c)$.

d. III is convex on $[c, d]$.

Programming Problems

LASSO: Code is all together

```python
# Brian Kang
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# make synthetic data
n = 500  # row
d = 1000  # column
k = 100  # ???
sd = 1  # standard normal sd
X = np.random.randn(n, d)  # sample from standard normal dist
error = np.random.randn(n)  # sample from standard normal dist
wj = np.empty(d)  # made up weights
for j in range(d):
    if j < k:
        wj[j] = j / k
    else:
        wj[j] = 0
Y = np.dot(X, wj) + error  # synthetic data


# get smallest lambda that shrinks all w to 0
def lambdaMax(X, Y):
    ybar = Y - np.mean(Y)
    return np.amax(2 * np.absolute(np.dot(X.T, ybar)))


# do coordinate descent algorithm for lasso
def coordDescentLasso(X, Y, w0, lmbda, tol):
    n, d = X.shape
    a = np.zeros(d)
    c = np.zeros(d)
    wNew = np.copy(w0)  # copy of initial weights, changing one vars doesn't affect the other
    doCoordDescent = True  # while not converged, do...
    while doCoordDescent:
        wOld = np.copy(wNew)  # after first loop, wNew is now wOld
        b = np.sum(Y - np.dot(X, wNew)) / n  # get b value
        for k in range(d):
            a[k] = 2 * np.sum(np.power(X[:, k], 2))
            resid = Y - (b + np.dot(X, wNew) - X[:, k] * wNew[k])  # subtract since we sum_{j!=k}
            c[k] = 2 * np.sum(X[:, k] * resid)
            # update weights
            if c[k] < -lmbda:
                wNew[k] = (c[k] + lmbda) / a[k]
            elif c[k] > lmbda:
                wNew[k] = (c[k] - lmbda) / a[k]
            else:
                wNew[k] = 0
```

```python
        doCoordDescent = (np.amax(np.absolute(wNew - wOld)) > tol)  # false if
    error < tolerance
    return wNew, b


def problem4():
    w0 = np.random.randn(d)  # make up starting weights
    # keep weights, b values
    keepW = []  # list, not np.array
    keepB = []

    # Compute lasso along regularization path
    lmbdaMax = lambdaMax(X, Y)  # Largest penalty
    # print(lmbdaMax)
    numlasso = 20  # number of lasso's to run. increase until regPath[numlasso]
    close to 0
    regPath = [lmbdaMax] * numlasso
    for i in range(numlasso):
        regPath[i] = regPath[i] / np.power(1.5, i)
    # print(regPath)
    for lmbda in regPath:
        w, b = coordDescentLasso(X, Y, w0, lmbda, 0.1)
        keepW.append(w)
        keepB.append(b)

    # a)
    keepFeat = []
    for w in keepW:
        keepFeat.append(np.sum(w > 0))  # count how many features kept

    plt.figure(0)
    plt.plot(regPath, keepFeat)
    plt.xlabel("$\lambda$")
    plt.ylabel("Number of Non-Zero Features")
    plt.xscale('log')
    plt.savefig('hw2_1.png')

    # b)
    FDR = []  # false discovery rate
    TPR = []  # true positive rate
    for w in keepW:
        FDR.append(np.sum(w[k:] > 0) / (d - k))
        TPR.append(np.sum(w[:k] > 0) / k)

    plt.figure(1)
    plt.scatter(FDR, TPR)
    plt.xlabel("False Discovery Rate")
    plt.ylabel("True Positive Rate")
    plt.savefig('hw2_2.png')


def problem5():
    df_train = pd.read_table("crime-train.txt")  # load datasets
    df_test = pd.read_table("crime-test.txt")

```

```
    d = 95  # number of features (columns)
    Xbody = df_train.iloc[:, 1:96].copy()
    X_train = Xbody.values  # get a copy
    # print(X_train)
    X_test = df_test.iloc[:, 1:96].copy().values
    Y_train = df_train['ViolentCrimesPerPop'].values  # convert to numpy array
    # print(Y_train)
    Y_test = df_test["ViolentCrimesPerPop"].values

    w0 = np.zeros(d)  # intial weights
    # keep weights, b values
    keepW = []  # list, not np.array
    keepB = []

    # Compute lasso along regularization path
    lmbdaMax = lambdaMax(X_train, Y_train)  # Largest penalty
    lmbda = 1  # temporary arbitrary number
    regPath = []
    i = 0
    while lmbda > 0.01:  # keep doing lasso until lambda <= 0.01
        lmbda = lmbdaMax / np.power(2, i)
        # print(lmbda, ", ", i)
        regPath.append(lmbda)
        w, b = coordDescentLasso(X_train, Y_train, w0, lmbda, 0.1)
        w0 = w  # use new weights as the weights for next lasso
        keepW.append(w)
        keepB.append(b)
        i += 1

    # a)
    keepFeat = []
    for w in keepW:
        keepFeat.append(np.sum(w > 0))  # count how many features kept

    plt.figure(2)
    plt.plot(regPath, keepFeat)
    plt.xlabel("$\lambda$")
    plt.ylabel("Number of Non-Zero Features")
    plt.xscale('log')
    plt.savefig('hw2_3.png')

    # b)
    # get coefficients of specific vars from the lassos
    vars = ["agePct12t29", "pctWSocSec", "pctUrban", "agePct65up", "householdsize"]
    for i in vars:
        indx = Xbody.columns.get_loc(i)
        getCoeff = [row[indx] for row in keepW]
        plt.figure(3)
        line, = plt.plot(regPath, getCoeff)
        line.set_label(i)

    plt.xlabel("$\lambda$")
    plt.ylabel("Coefficients")
    plt.xscale('log')
```

```
    plt.legend()
    plt.savefig('hw2_4.png')

    # c)
    # get squared errors from the lassos
    errorTrain = [np.mean(np.square(np.dot(X_train, w) + b - Y_train)) for w, b in zip(keepW, keepB)]
    errorTest = [np.mean(np.square(np.dot(X_test, w) + b - Y_test)) for w, b in zip(keepW, keepB)]

    plt.figure(4)
    plt.plot(regPath, errorTrain, label="train")
    # line1.set_label("Train")
    plt.plot(regPath, errorTest, label="test")
    # line2.set_label("Test")
    plt.xlabel("$\lambda$")
    plt.ylabel("MSE")
    plt.xscale('log')
    plt.legend()
    plt.savefig('hw2_5.png')

    # d)
    # print(regPath)
    indx = 4  # index where lambda approx = 30
    getCoeff = [row[indx] for row in keepW]  # the weights
    print("Most Positive Lasso Coefficient: ", np.max(getCoeff))
    # Out: "Most Positive Lasso Coefficient:  0.005256729455865754"
    maxindx = np.argmax(getCoeff)
    print("Corresponding Variable:", Xbody.columns[maxindx])
    # Out: "perCapInc"
    print("Most Negative Lasso Coefficient: ", np.min(getCoeff))
    # Out: "Most Negative Lasso Coefficient:  0.0"
    minindx = np.argmin(getCoeff)
    print("Corresponding Variable:", Xbody.columns[minindx])
    # Out: "population"


problem4()
problem5()
```

Plots and solutions are below!

A.4

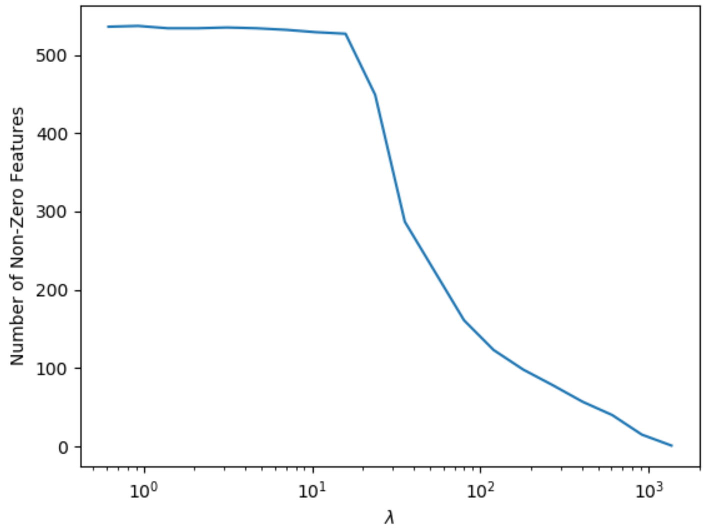

Figure 1: Part a code is above.

a.

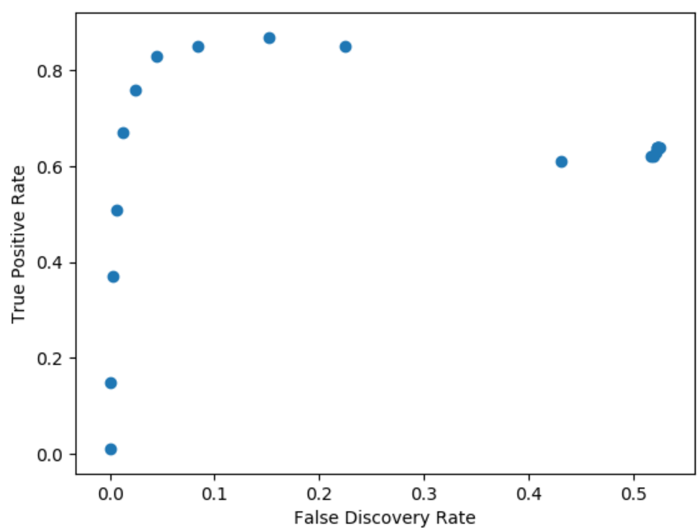

Figure 2: Part b code is above.

b.

c. We observe that as lambda decreases by linear fractions (from right to left in figure 1), we move from left to right in FDR v.s. TPR (figure 2). The sharp decrease in figure 1 and sharp increase in figure 2 towards ideal (upper left-hand corner) are correlated to each other, as the lasso is at work with feature selection with respect to various regularization lambda values. But as we go further, FDR increases rapidly and TPR even decreases, which is reasonable since the number of non-zero features kept increase in a decreasing rate.

A.5

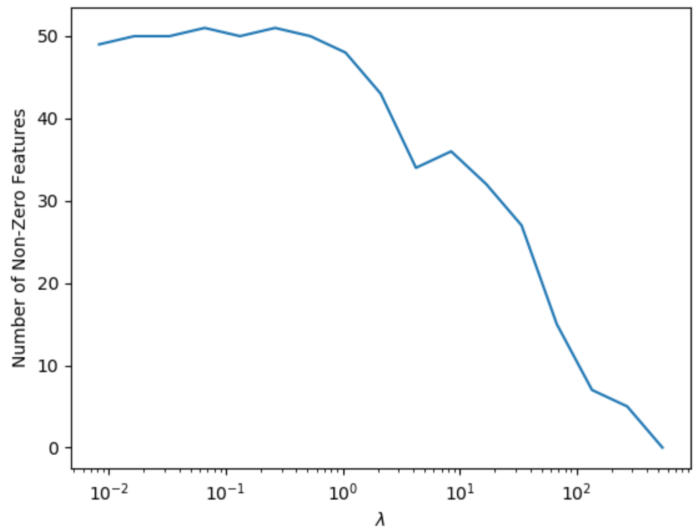

Figure 3: Part a code is above.

a.

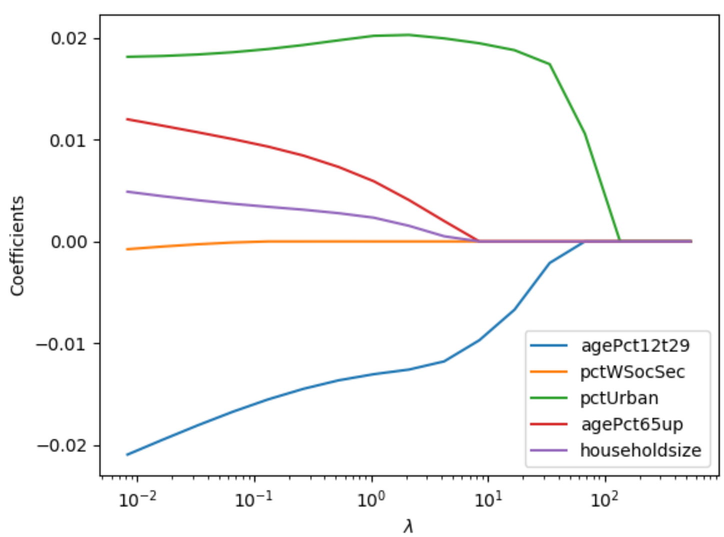

Figure 4: Part b code is above.

b.

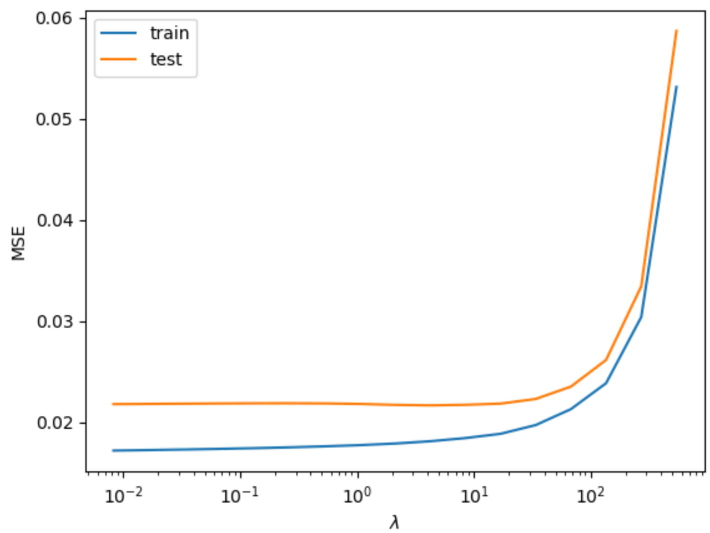

Figure 5: Part c code is above.

c.

d. We found out that the variable with the most positive lasso coefficient = 0.00525673 is **perCapInc**, and the variable with the most "negative" lasso coefficient = 0.0 is **population**. Since we are predicting Per Capita Violent Crimes, it is reasonable to say that Per Capita Income is a important feature because usually there is a correlation between crime rates and income. But when it comes to population, the lasso shrunk the coefficient to zero because Per Capita Violent Crimes is probably calculated by dividing population. The machine doesn't know this so population is treated as an insignificant feature for prediction.

e. This method of reasoning is incorrect and it is known as Reverse Causality. Just because there is a correlation, we cannot conclude with a solution resulting in "if X, then Y" because there is always the case of "if Y, then X" for correlations. And in cases like this (and the fire truck/burning building example) this is what is happening. For causal inference we need a clear experiment design and domain knowledge to know that "fire trucks don't cause fire."

LOGIT: Code is all together

```python
# Brian Kang
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST


def load_dataset():
    mndata = MNIST('.\data')
    mndata.gz = True
    X_train, labels_train = map(np.array, mndata.load_training())
    X_test, labels_test = map(np.array, mndata.load_testing())
    X_train = X_train / 255.0
    X_test = X_test / 255.0

    # get index array of 0 or 1
    keep_train = np.bitwise_or(labels_train == 2, labels_train == 7)
    keep_test = np.bitwise_or(labels_test == 2, labels_test == 7)

    # delete columns that don't give 2 or 7 response
    X_train = X_train[keep_train]
    labels_train = labels_train[keep_train].astype(int)
    X_test = X_test[keep_test]
    labels_test = labels_test[keep_test].astype(int)

    # binary classify to -1, 1
    labels_train[labels_train == 2] = -1
    labels_train[labels_train == 7] = 1
    labels_test[labels_test == 2] = -1
    labels_test[labels_test == 7] = 1
    return X_train, labels_train, X_test, labels_test


x_train, bin_train, x_test, bin_test = load_dataset()  # load in data


# calculate mu
def mu(w, b, X, Y):
    return 1 / (1 + np.exp(-Y * (b + X.T @ w)))


# get the cost function J
def J(w, b, X, Y, lmbda):
    _, n = X.shape
    return np.sum(np.log(1 / mu(w, b, X, Y))) / n + lmbda * np.linalg.norm(w) ** 2


# gradient of w on J (dell w J)
def dwJ(w, b, X, Y, lmbda):
    d, n = X.shape
    mu_i = mu(w, b, X, Y)
    # do matrix algebra much faster
    # https://stackoverflow.com/questions/26089893/understanding-numpys-einsum
```

```python
    return np.einsum("i, i, di -> d", (1 - mu_i), -Y, X) / n + 2 * lmbda * w


# gradient of b on J (dell b J)
def dbJ(w, b, X, Y):
    d, n = X.shape
    mu_i = mu(w, b, X, Y)
    # do element wise multiplication faster
    return np.einsum("i, i -> ", (1 - mu_i), -Y) / n


# gradient descent
def gd(w0, b0, X, Y, lmbda, stepSize, iter):
    # step size = learning rate
    keepJ = []
    # start will all zeros
    keepW = np.zeros((iter, len(w0)))
    keepB = np.zeros(iter)
    keepW[0] = w0  # 0th row is the starting weights
    keepB[0] = b0  # 0th row is the starting intercept
    keepJ.append(J(w0, b0, X, Y, lmbda))  # keep the 0th cost J
    for i in range(1, iter):
        # new = old - step size * gradient
        keepW[i] = keepW[i - 1] - stepSize * dwJ(keepW[i - 1], keepB[i - 1], X, Y, lmbda)
        # print(i,":")
        # print("keepW: ", keepW)
        keepB[i] = keepB[i - 1] - stepSize * dbJ(keepW[i - 1], keepB[i - 1], X, Y)
        # print("keepB: ", keepB)
        keepJ.append(J(keepW[i], keepB[i], X, Y, lmbda))
        # print("keepJ: ", keepJ)
        i += 1
    return keepW, keepB, keepJ


# classification error
def classifier(w, b, X, Y):
    d, n = X.shape
    levelx = np.sign(b + X.T @ w) > 0  # classify to 1 or -1
    levely = Y == 1
    return 1 - np.sum(np.equal(levelx, levely) / n)


def sgd(w0, b0, X, Y, lmbda, stepSize, iter, batchSize):
    keepJ = []
    # start will all zeros
    keepW = np.zeros((iter, len(w0)))
    keepB = np.zeros(iter)
    keepW[0] = w0  # 0th row is the starting weights
    keepB[0] = b0  # 0th row is the starting intercept
    keepJ.append(J(w0, b0, X, Y, lmbda))  # keep the 0th cost J
    for i in range(1, iter):
        batch = np.random.randint(0, len(Y), batchSize)  # random index selection
```

```python
        # new = old - step size * gradient
        keepW[i] = keepW[i - 1] - stepSize * dwJ(keepW[i - 1], keepB[i - 1], X[:, batch], Y[batch], lmbda)
        keepB[i] = keepB[i - 1] - stepSize * dbJ(keepW[i - 1], keepB[i - 1], X[:, batch], Y[batch])
        keepJ.append(J(keepW[i], keepB[i], X, Y, lmbda))
        i += 1
    return keepW, keepB, keepJ


def problem6b():
    d, _ = x_train.T.shape
    w0 = np.zeros(d)
    b0 = 0
    lmbda = 0.1
    iter = 100
    trainW, trainB, trainJ = gd(w0, b0, x_train.T, bin_train, lmbda, 0.05, iter)
    testW, testB, testJ = gd(w0, b0, x_test.T, bin_test, lmbda, 0.05, iter)

    plt.figure(0)
    plt.plot(range(0, iter), trainJ, label="train")
    plt.plot(range(0, iter), testJ, label="test")
    plt.xlabel("Iteration")
    plt.ylabel("Cost J(w,b)")
    plt.legend()
    plt.savefig('hw2_6.png')

    train_error = [classifier(w, b, x_train.T, bin_train) for w, b in zip(trainW, trainB)]
    test_error = [classifier(w, b, x_test.T, bin_test) for w, b in zip(testW, testB)]
    plt.figure(1)
    plt.plot(range(0, iter), train_error, label="train")
    plt.plot(range(0, iter), test_error, label="test")
    plt.xlabel("Iteration")
    plt.ylabel("Misclassification Error")
    plt.legend()
    plt.savefig('hw2_7.png')


def problem6c():
    d, _ = x_train.T.shape
    w0 = np.zeros(d)
    b0 = 0
    lmbda = 0.1
    iter = 100
    batch = 1
    trainW, trainB, trainJ = sgd(w0, b0, x_train.T, bin_train, lmbda, 0.05, iter, batch)
    testW, testB, testJ = sgd(w0, b0, x_test.T, bin_test, lmbda, 0.05, iter, batch)

    plt.figure(2)
    plt.plot(range(0, iter), trainJ, label="train")
    plt.plot(range(0, iter), testJ, label="test")
```

```
    plt.xlabel("Iteration")
    plt.ylabel("Cost J(w,b)")
    plt.legend()
    plt.savefig('hw2_8.png')

    train_error = [classifier(w, b, x_train.T, bin_train) for w, b in zip(trainW
    , trainB)]
    test_error = [classifier(w, b, x_test.T, bin_test) for w, b in zip(testW,
    testB)]
    plt.figure(3)
    plt.plot(range(0, iter), train_error, label="train")
    plt.plot(range(0, iter), test_error, label="test")
    plt.xlabel("Iteration")
    plt.ylabel("Misclassification Error")
    plt.legend()
    plt.savefig('hw2_9.png')


def problem6d():
    d, _ = x_train.T.shape
    w0 = np.zeros(d)
    b0 = 0
    lmbda = 0.1
    iter = 100
    batch = 100
    trainW, trainB, trainJ = sgd(w0, b0, x_train.T, bin_train, lmbda, 0.05, iter
    , batch)
    testW, testB, testJ = sgd(w0, b0, x_test.T, bin_test, lmbda, 0.05, iter,
    batch)

    plt.figure(4)
    plt.plot(range(0, iter), trainJ, label="train")
    plt.plot(range(0, iter), testJ, label="test")
    plt.xlabel("Iteration")
    plt.ylabel("Cost J(w,b)")
    plt.legend()
    plt.savefig('hw2_10.png')

    train_error = [classifier(w, b, x_train.T, bin_train) for w, b in zip(trainW
    , trainB)]
    test_error = [classifier(w, b, x_test.T, bin_test) for w, b in zip(testW,
    testB)]
    plt.figure(5)
    plt.plot(range(0, iter), train_error, label="train")
    plt.plot(range(0, iter), test_error, label="test")
    plt.xlabel("Iteration")
    plt.ylabel("Misclassification Error")
    plt.legend()
    plt.savefig('hw2_11.png')


problem6b()
problem6c()
problem6d()
```

Plots and solutions are below!

A.6

a. *Proof.*

$$\begin{aligned}
\nabla_w J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} \frac{1}{1+\exp(-y_i(b+x_i^T w))} * \exp(-y_i(b+x_i^T w)) * -y_i x_i + 2\lambda w \\
&= \frac{1}{n}\sum_{i=1}^{n} \mu_i(w,b) * \frac{1}{\mu_i(w,b)-1} * -y_i x_i + 2\lambda w \\
&= \frac{1}{n}\sum_{i=1}^{n} \frac{\mu_i(w,b)}{\mu_i(w,b)-1} * -y_i x_i + 2\lambda w
\end{aligned}$$

Or equivalently, we can get,

$$\begin{aligned}
\nabla_w J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} \frac{\exp(-y_i(b+x_i^T w))}{1+\exp(-y_i(b+x_i^T w))} * -y_i x_i + 2\lambda w \\
&= \frac{1}{n}\sum_{i=1}^{n} \frac{1+\exp(-y_i(b+x_i^T w))-1}{1+\exp(-y_i(b+x_i^T w))} * -y_i x_i + 2\lambda w \\
&= \frac{1}{n}\sum_{i=1}^{n} (1-\frac{1}{1+\exp(-y_i(b+x_i^T w))}) * -y_i x_i + 2\lambda w \\
&= \frac{1}{n}\sum_{i=1}^{n} (1-\mu_i(w,b)) * -y_i x_i + 2\lambda w
\end{aligned}$$

$$\begin{aligned}
\nabla_b J(w,b) &= \frac{1}{n}\sum_{i=1}^{n} \frac{1}{1+\exp(-y_i(b+x_i^T w))} * \exp(-y_i(b+x_i^T w)) * -y_i \\
&= \frac{1}{n}\sum_{i=1}^{n} \frac{\mu_i(w,b)}{\mu_i(w,b)-1} * -y_i \\
&= \frac{1}{n}\sum_{i=1}^{n} (1-\mu_i(w,b)) * -y_i
\end{aligned}$$

□

b. Part b plots for gradient descent.

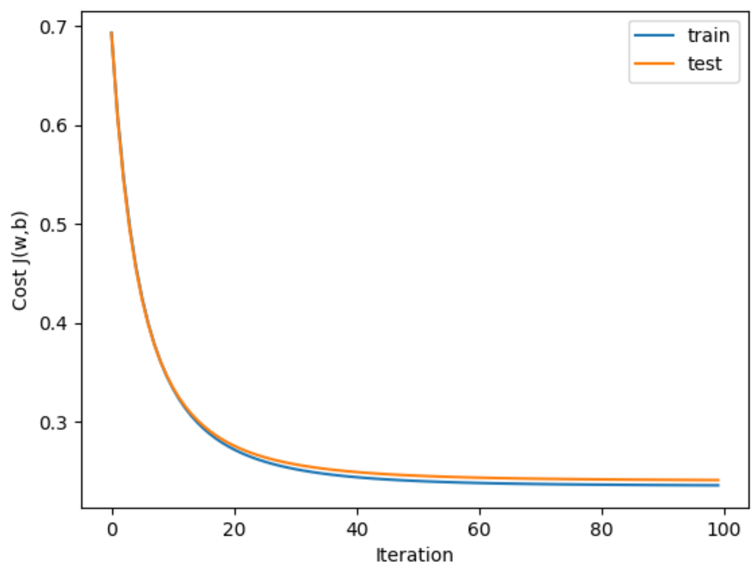

Figure 6: Part i (J(w,b) Cost) code is above.

i

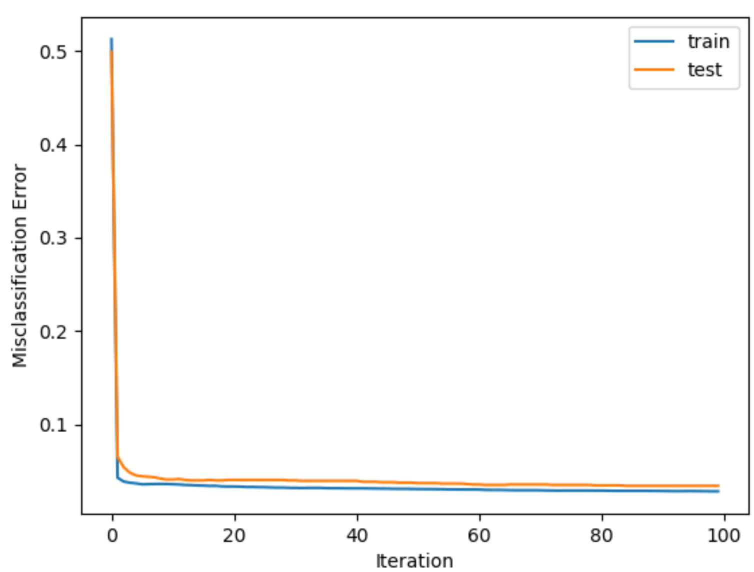

Figure 7: Part ii (Misclassification Error) code is above.

ii

c. Part c plots for stochastic gradient descent with batch size=1.

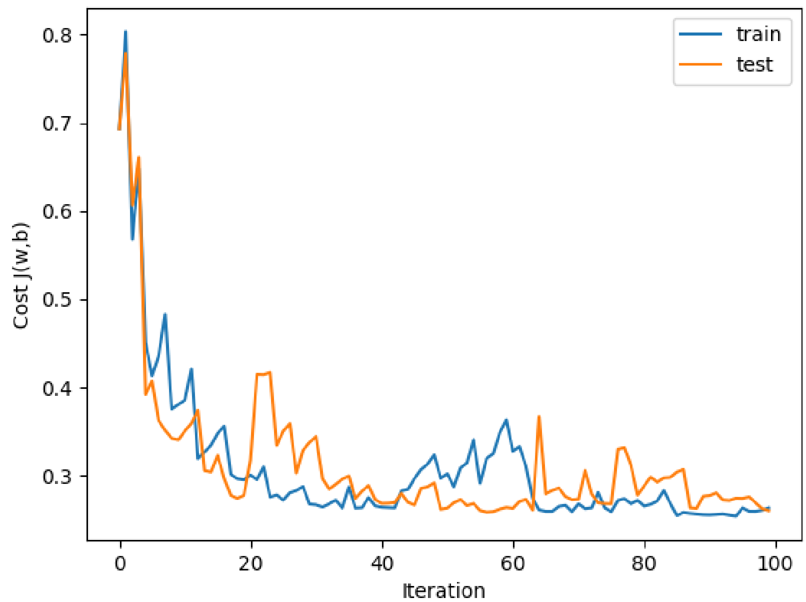

Figure 8: J(w,b) Cost code is above.

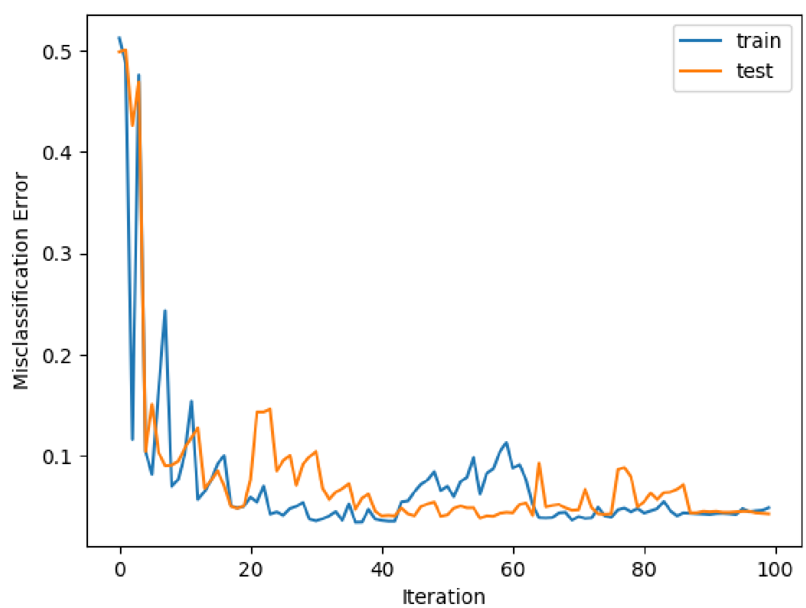

Figure 9: Misclassification Error code is above.

d. Part d plots for stochastic gradient descent with batch size=100.

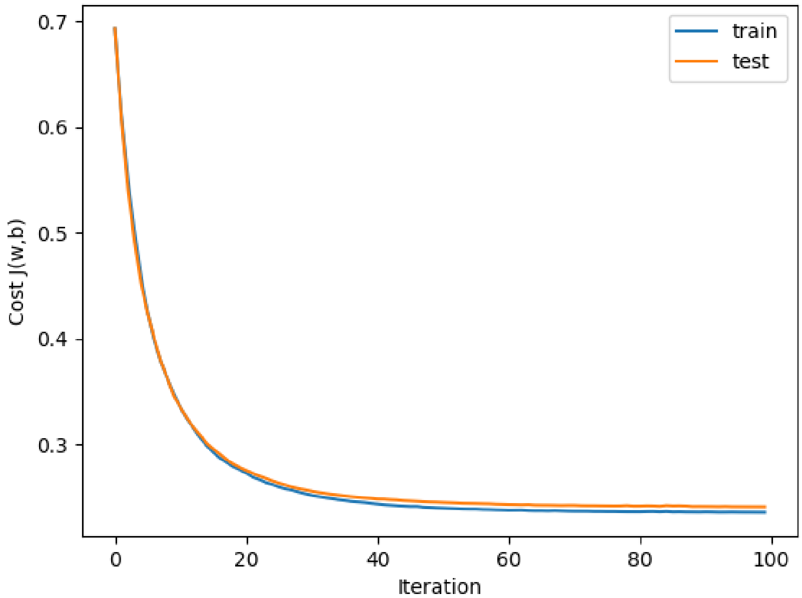

Figure 10: J(w,b) Cost code is above.

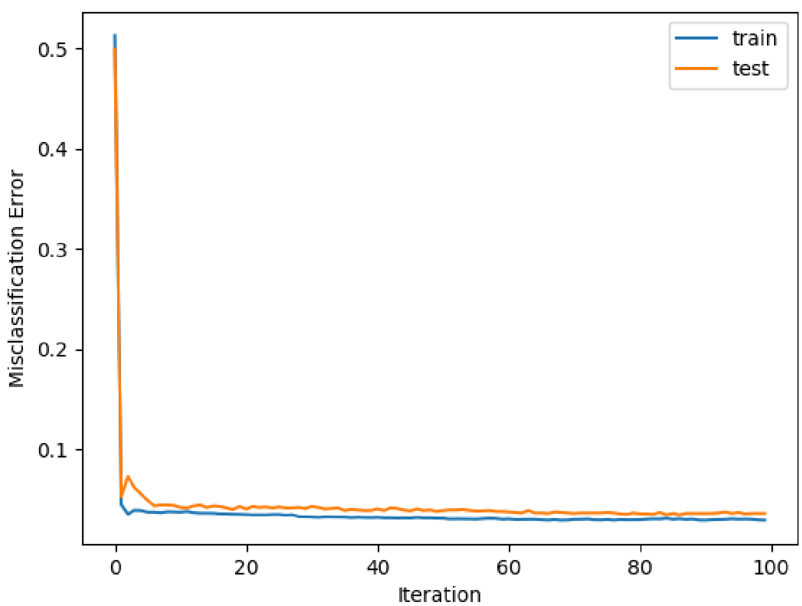

Figure 11: Misclassification Error code is above.