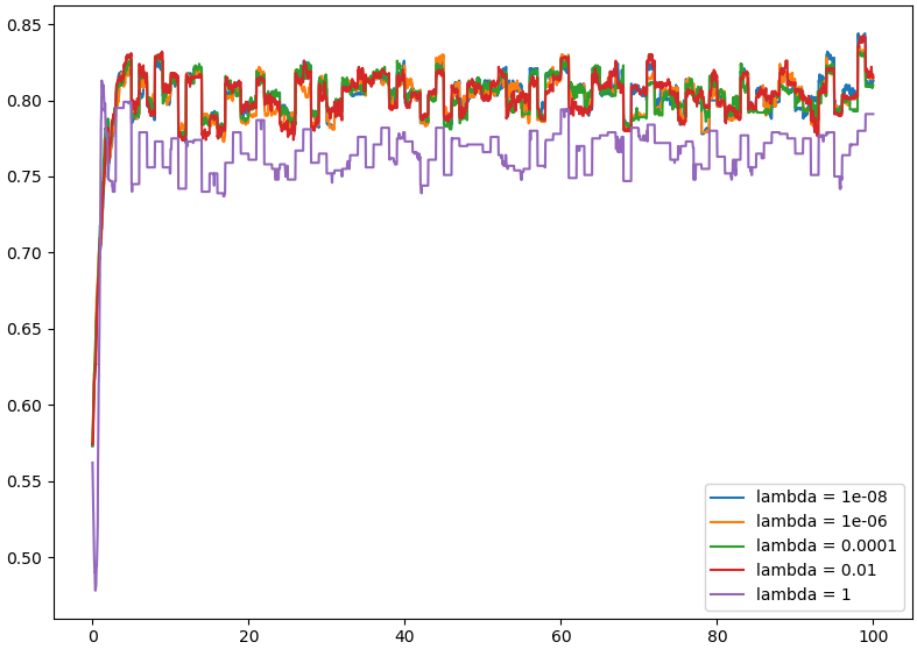
STUDENT

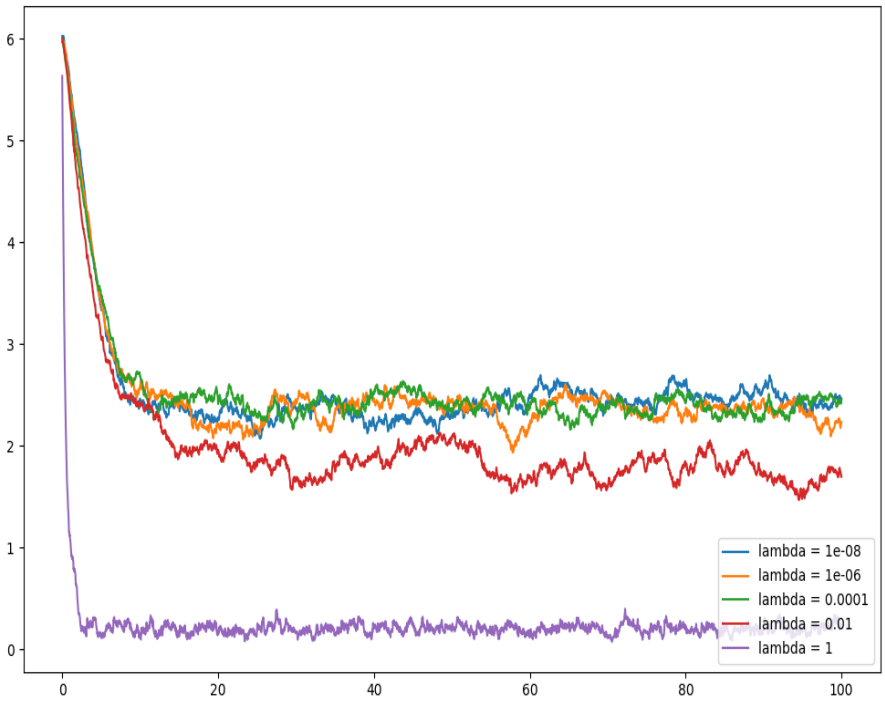
Huajian Lin

AUTOGRADER SCORE

81.51 / 100.0



I use 100 seasons rather than minimum 50 suggested on the HW. But from 50 to 100 seasons, it look pretty identical to previous trend, so it just serves as a safety check.



I use 100 seasons rather than minimum 50 suggested on the HW. But from 50 to 100 seasons, it look pretty identical to previous trend, so it just serves as a safety check.

Best estimate of regularization constant is **0.0001.** I pick this by setting up 8 different lambda values from 10^-8 to 10^-1. Through experiment I only kept 5 and lambda = 0.0001 gives the best accuracy base on the held out validation set.

For learning rate, it is **n/(s+m)** where s is the season number and **n = 100, m = 20000.** I start by picking n = 1 and m = 100 but quickly release that this rate learning is to high and cause accuracy to jump up and down. So I increase m to make learning rate smaller. However, I encounter another case where when learning rate is to small, I never reach best accuracy. Hence the final learning rate is a balance of increase m and n.

Training of an SVM:

```
#loop to train models build with different lambdas
for season in range(1, seasons+1):
    learnRate = 1/(0.1*season+200)
    trainFVDiv, testHeldout, trainLabsDiv, testHeldoutLabels = [],[], [], []
    dataProcess.getRandomN(accTestNumber, testHeldout, testHeldoutLabels,
trainFVDiv, trainLabsDiv, trainModelFV, trainModelLabels)
    for step in range(1, stepsPerSeason+1):
        #update each model use
        for lam in range(0, len(lambdas)):
            x,y = dataProcess.getRandomSample(trainFVDiv, trainLabsDiv)
            gradiantDecent(x, y, aVectors, bVector, lambdas, learnRate, lam)
            if(step % confirmAccSteps == 0):
                tempAccs[lam].append(getTempAcc(aVectors[lam], bVector[lam],
testHeldout, testHeldoutLabels))
                coefVectors[lam].append(np.sum(np.abs(aVectors[lam])))
```

SGD:

```
#calcualte gradiant and update
def gradiantDecent(dataSet, labels, a, b, lambdas, learnRate, lam):
    cost = getCost(x, y, a[lam], b[lam], lambdas[lam])
    if(cost != 0):
        a[lam] -= np.dot(learnRate,(np.dot(a[lam],lambdas[lam])-np.dot(x,y)))
        b[lam] += learnRate*y
    else:
        a[lam] -= np.dot(learnRate,np.dot(a[lam],lambdas[lam]))
```

Cost Function:

```
#calculate the cost
def getCost(x, y, a, b, lam):
    a = np.array(a)
    cost = y*(np.dot(a,x)+b)+(lam*np.dot(a,a.T))/2
    return max(0, 1-cost)
```

Test SVM

```
#calculate accuracy for given dataset
def getTempAcc(a, b, testSet, testLabels):
    correct = 0
    for i in range(0, len(testSet)):
        ytemp = np.dot(a,testSet[i]) + b
        if(ytemp*testLabels[i]>=0):
            correct += 1
    return correct / len(testSet)
```

Complete SVM:

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

#calculate the cost
def getCost(x, y, a, b, lam):
    a = np.array(a)
    cost = y*(np.dot(a,x)+b)+(lam*np.dot(a,a.T))/2
    return max(0, 1-cost)

#calculate accuracy for given dataset
def getTempAcc(a, b, testSet, testLabels):
    correct = 0
    for i in range(0, len(testSet)):
        ytemp = np.dot(a,testSet[i]) + b
        if(ytemp*testLabels[i]>=0):
            correct += 1
    return correct / len(testSet)

#calcualte gradiant and update
def gradiantDecent(dataSet, labels, a, b, lambdas, learnRate, lam):
    cost = getCost(x, y, a[lam], b[lam], lambdas[lam])
    if(cost != 0):
        a[lam] -= np.dot(learnRate,(np.dot(a[lam],lambdas[lam])-np.dot(x,y)))
        b[lam] += learnRate*y
    else:
        a[lam] -= np.dot(learnRate,np.dot(a[lam],lambdas[lam]))

#defien variables for configuration and storage
continousFeatureIndexs = [0,2,4,10,11,12]
trainFileName, testFileName = 'train.txt', 'test.txt'
trainFV, testFV, trainLabels, testLabels = [], [], [], []

dataProcess.processData(trainFileName, trainFV, trainLabels,
continousFeatureIndexs)
dataProcess.processData(testFileName, testFV, testLabels,
continousFeatureIndexs)

trainFV = dataProcess.dataNormalization(trainFV)
testFV = dataProcess.dataNormalization(testFV)

#define constants for training SVM, this SVM use sign(ax + b) to make
```

```
prediction
lambdas = [0.00000001, 0.000001, 0.0001, 0.0007, 1]
aVectors = [[1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1,
1, 1, 1, 1], [1, 1, 1, 1, 1, 1]]
bVector = [1,1,1,1,1]
batchSize, seasons, stepsPerSeason, accTestNumber, confirmAccSteps = 1, 100,
1000, 400, 30
tempAccs, coefVectors = [[],[],[],[],[]], [[],[],[],[],[]]

#split the train data to 90-10 for training and validating
trainModelFV, testModelFV, trainModelLabels, testModelLabels = [], [], [], []
ptest = 0.1
dataProcess.splitData(ptest, testModelFV, testModelLabels, trainModelFV,
trainModelLabels, trainFV, trainLabels)

#loop to train models build with different lambdas
for season in range(1, seasons+1):
    learnRate = 1/(0.1*season+200)
    trainFVDiv, testHeldout, trainLabsDiv, testHeldoutLabels = [],[], [], []
    dataProcess.getRandomN(accTestNumber, testHeldout, testHeldoutLabels,
trainFVDiv, trainLabsDiv, trainModelFV, trainModelLabels)
    for step in range(1, stepsPerSeason+1):
        #update each model use
        for lam in range(0, len(lambdas)):
            x,y = dataProcess.getRandomSample(trainFVDiv, trainLabsDiv)
            gradiantDecent(x, y, aVectors, bVector, lambdas, learnRate, lam)
            if(step % confirmAccSteps == 0):
                tempAccs[lam].append(getTempAcc(aVectors[lam], bVector[lam],
testHeldout, testHeldoutLabels))
                coefVectors[lam].append(np.sum(np.abs(aVectors[lam])))

#plot  accuracy
for i in range(0, len(tempaccs)):
    x = np.linspace(0,seasons,len(tempaccs[i]))
    plt.plot(x, tempaccs[i], label=f'lambda = {lambdas[i]}')
plt.legend(loc='lower right')
plt.show()

#plot aVector magnitude
for i in range(0, len(coefvectors)):
    x = np.linspace(0,seasons,len(coefvectors[i]))
    plt.plot(x, coefvectors[i], label=f'lambda = {lambdas[i]}')
plt.legend(loc='lower right')
plt.show()
```

```
#train using all training set
finalA = [1,1,1,1,1,1]
bestLam = np.argmax(np.mean(tempAccs, axis = 1))
finalB = 1
outputFile = 'submission.txt'

for season in range(1, seasons+1):
    learnRate = 1/(0.01*season+200)
    for step in range(1, stepsPerSeason+1):
        x,y = dataProcess.getRandomSample(trainFV, trainLabels)
        cost = getCost(x, y, finalA, finalB, bestLam)
        if(cost != 0):
            finalA -= np.dot(learnRate,(np.dot(finalA,bestLam)-np.dot(x,y)))
            finalB += learnRate*y
        else:
            finalA -= np.dot(learnRate,np.dot(finalA,bestLam))

#predict unseen data
with open(outputFile, 'w') as file:
    for i in range(0, len(testFV)):
        label = np.dot(finalA,testFV[i]) + finalB
        if(label <= 0):
            file.write('<=50K\n')
        else:
            file.write('>50K\n')
```

Data Processing:

```
import numpy as np
import random

#take a txt file name, return only continuous variable as part of feature
vector
def processData(fileName, featureVector, labels, continousFeatureIndexs):
    with open(fileName) as file:
        data = file.readlines()
        for line in data:
            features = line.split(',')
            featureVector.append([float(features[x]) for x in
range(0,len(features)) if x in continousFeatureIndexs])
            label = features[len(features)-1].lstrip()
            if(label[0] == '<'):
                labels.append(-1)
```

```
            else:
                labels.append(1)

#return normalized data
def dataNormalization(featureVector):
    return (featureVector - np.mean(featureVector, axis = 0)) /
np.std(featureVector, axis=0)

#split data into test and train with probability of pTest picking a datapoint
for testset
def splitData(pTest, test, testLables, train, trainLables, dataSet, labels):
    for i in range(0, len(dataSet)):
        #random.random() return a random number between (0,1]
        randPTest = random.random()
        if(randPTest <= pTest):
            test.append(dataSet[i])
            testLables.append(labels[i])
        else:
            train.append(dataSet[i])
            trainLables.append(labels[i])

#randomly split data so that n data point will be in test, and the rest will be
in train
def getRandomN(n, test, testLabels, train, trainLabels, dataSet, labels):
    pickForTest = random.sample(range(0,len(dataSet)),n)
    for i in range(0, len(dataSet)):
        if(i in pickForTest):
            test.append(dataSet[i])
            testLabels.append(labels[i])
        else:
            train.append(dataSet[i])
            trainLabels.append(labels[i])

#return a random sample and label
def getRandomSample(dataSet, labels):
    randomIndex = random.sample(range(0, len(dataSet)), 1)[0]
    return (dataSet[randomIndex], labels[randomIndex])
```