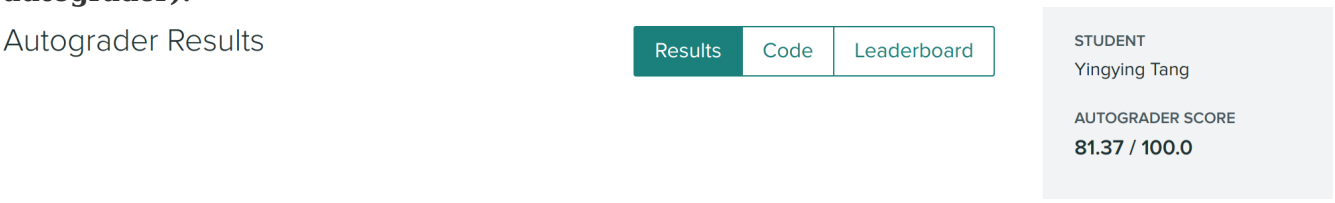
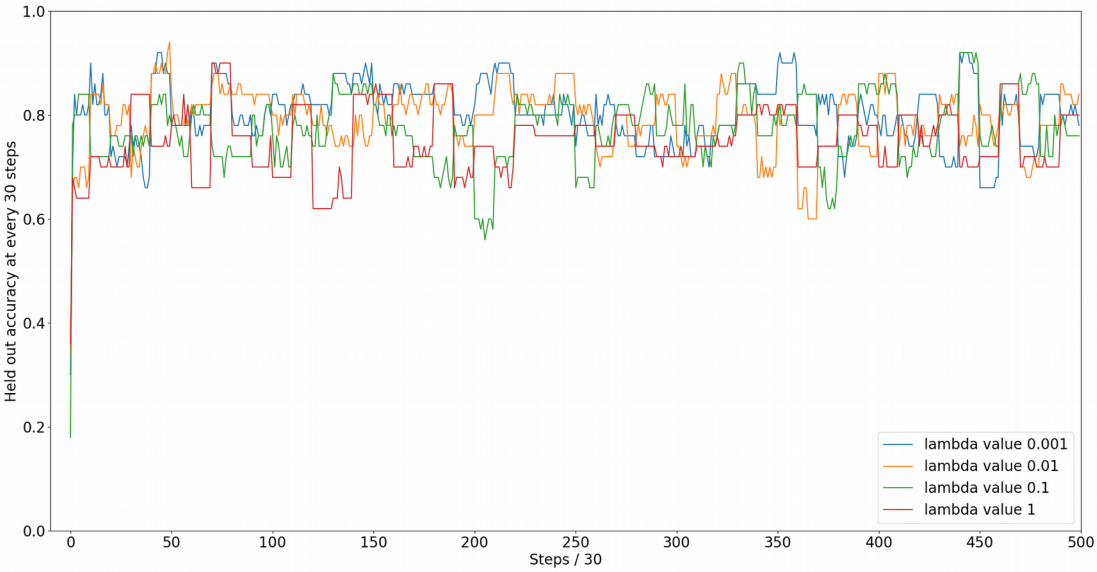
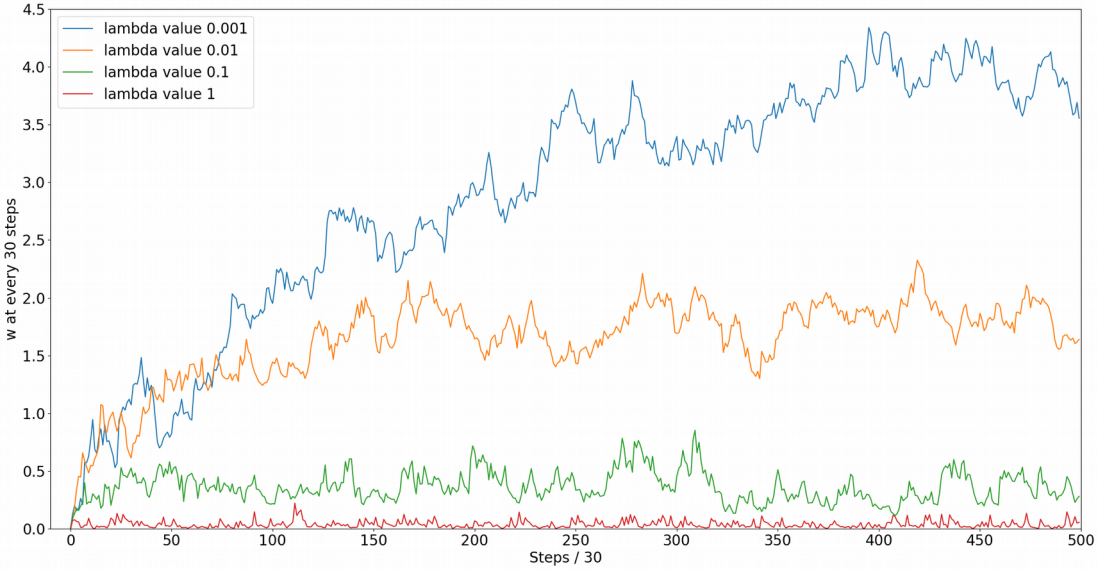
Page 1 A screenshot of your best accuracy on the test set (retrieve this from the autograder).



Page 2 A plot of the validation accuracy every 30 steps, for each value of the regularization constant. You should plot the curves for all regularization constants in the same plot using different colors with a label showing the corresponding values.



Page 3 A plot of the magnitude of the coefficient vector every 30 steps, for each value of the regularization constant. You should plot the curves for all regularization constants in the same plot using different colors with a label showing the corresponding values.



Page 4 Your estimate of the best value of the regularization constant, together with a brief description of why you believe that is a good value. What was your choice for the learning rate and why did you choose it?

The best value of the regularization constant is 0.001. Have trained and tested the dataset with different regularization constant values, including 0.001, 0.01, 0.1, 1. As seen from the results:

- Accuracy of the valuation dataset is 0.8043676069153776 with lambda value 0.001 and m value 0
- Accuracy of the valuation dataset is 0.8043676069153776 with lambda value 0.01 and m value 0
- Accuracy of the valuation dataset is 0.7791173794358508 with lambda value 0.1 and m value 0
- Accuracy of the valuation dataset is 0.7618289353958144 with lambda value 1 and m value 0

we achieved the maximum accuracy with lambda value 0.01 and 0.001. Since with larger value of lambda, the penalty is more, which could cause under fitting with less contribution of the features.
Therefore, the best value of the regulation constant is 0.001.

The learning rate is 1 / (0.001 * s + 50). Similar to the selection of lambda value, different learning rate values, 1/50 and 1 / (0.001 * s + 50), are used for training and testing. The learning rate of 1 / (0.001 * s + 50) achieved a higher accuracy.

Page 5 A screenshot of your code.

- **Training of an SVM (these are part of the codes. Function details are in page 6 complete codes)**

```
    for lambdaVal in lambdaVals:
        self.a = np.zeros(D)
        self.b = 0
        accuracy = []
        w = []
        for season in range(self.Seasons):
            season += 1
            lr = 1 / ((m * season) + n)
            # Take out 50 examples for testing at every 30 step
            combinedXY = np.concatenate((X, np.matrix(Y).T), axis=1)
            accuracyData, trainData = splitDataseteByNumber(combinedXY, 50)
            accuracyX = accuracyData[:, :-1]
            accuracyY = np.array(accuracyData[:, -1]).flatten()
            trainDataX = trainData[:, :-1]
            trainDataY = np.array(trainData[:, -1]).flatten()
            # gradient descent
            for step in range(self.Steps):
                if (step % self.evaluationStep == 0):
                    accuracy.append(self.score(accuracyX, accuracyY))
                    w.append(self.a.dot(self.a))
                k = np.random.choice(N - 50, 1)[0]
                Xk = np.array(trainDataX[k, :]).flatten()
                yk = trainDataY[k]
                margins = yk * self.linear_function(Xk)
                if (margins >= 1):
                    pa = lambdaVal * self.a
                    pb = 0
                else:
                    pa = lambdaVal * self.a - yk * Xk
                    pb = -yk
                self.a -= lr * pa
                self.b -= lr * pb
```

- **Testing of an SVM (these are part of the codes. Function details are in page 6 complete codes)**

```
    def testSetScore(self, lambdaVal, trainDataset, testDataset, m, n):
     # Select the columns with continuous values
    trainX = trainDataset[:, [0, 2, 4, 10, 11, 12]].astype(np.float)
    testX = testDataset[:, [0, 2, 4, 10, 11, 12]].astype(np.float)
    # Normalization of the dataset features
    trainDataX = normalization(trainX)
    testX = normalization(testX)
    # Update class values
    trainDataY = np.where(trainDataset[:, -1] == ' <=50K', -1, 1)
    N, D = trainDataX.shape
    self.a = np.zeros(D)
    self.b = 0
    for season in range(self.Seasons):
        season += 1
        lr = 1 / ((m * season) + n)
        # gradient descent
        for step in range(self.Steps):
            k = np.random.choice(N - 50, 1)[0]
            Xk = np.array(trainDataX[k, :]).flatten()
            yk = trainDataY[k]
            margins = yk * self.linear_function(Xk)
            if (margins >= 1):
                pa = lambdaVal * self.a
                pb = 0
            else:
                pa = lambdaVal * self.a - yk * Xk
                pb = -yk
            self.a -= lr * pa
            self.b -= lr * pb
    # Predict the test dataset
    predictY = np.where(self.linear_function(testX) >= 0, ">50K", "<=50K")
    np.savetxt('submission.txt', predictY, fmt='%s')
```

Page 6+ All code should be attached at the end of the pdf. There is no limit to the number of pages required for full code print.

```
from __future__ import print_function, division
from future.utils import iteritems
from builtins import range, input
import matplotlib
import matplotlib.pyplot as plt
from datetime import datetime
import numpy as np
from numpy import genfromtxt
import scipy
def loadCsv(filename):
    dataset = genfromtxt(filename, delimiter=',', dtype='str')
    return dataset
def splitDataseByNumber(dataset, number):
    arr = np.arange(dataset.shape[0])
    np.random.shuffle(arr)
    trainInd = arr[:number]
    testInd = arr[number:]
    trainSet, testSet = dataset[trainInd, :], dataset[testInd, :]
    return [trainSet, testSet]
def splitDatase(dataset, splitRatio):
    arr = np.arange(dataset.shape[0])
    np.random.shuffle(arr)
    ind = int(arr.shape[0] * splitRatio)
    trainInd = arr[:ind]
    testInd = arr[ind:]
    trainSet, testSet = dataset[trainInd, :], dataset[testInd, :]
    return [trainSet, testSet]
def normalization(dataset):
    dataset -= np.mean(dataset, axis=0)
    dataset /= np.std(dataset, axis=0)
    return dataset
def selectFeatureDataset(trainSet, validationSet):
    # Select the columns with continuous values
    trainX = trainSet[:, [0, 2, 4, 10, 11, 12]].astype(np.float)
    validationX = validationSet[:, [0, 2, 4, 10, 11, 12]].astype(np.float)
    # Normalization of the dataset features
    trainX = normalization(trainX)
    validationX = normalization(validationX)
    return [trainX, validationX]
def arrangeDataset(dataset, splitRatio):
    trainSet, validationSet = splitDatase(dataset, splitRatio)
    [trainX, validationX] = selectFeatureDataset(trainSet, validationSet)
    # Update class values
    trainY = np.where(trainSet[:, -1] == ' <=50K', -1, 1)
    validationY = np.where(validationSet[:, -1] == ' <=50K', -1, 1)
    return [trainX, trainY, validationX, validationY]
class SVM:
    def __init__(self, seasons=50, steps=300):
        self.Seasons = seasons
        self.Steps = steps
    def plotResult(self, accuracies, ws):
        measures = int(self.Seasons * self.Steps / self.evaluationStep)
        plt.figure(0)
        for i in range(0, len(accuracies)):
            plt.plot(accuracies[i])
        plt.ylim(0, 1)
        plt.xlim(-10, measures)
        plt.xticks(np.arange(0, measures + 1, 50), fontsize = 20)
        plt.yticks(fontsize=20)
        plt.legend(['lambda value 0.001', 'lambda value 0.01', 'lambda value 0.1', 'lambda value 1'],
loc='lower right', fontsize = 20)
        plt.xlabel("Steps / 30", fontsize = 20)
        plt.ylabel("Held out accuracy at every 30 steps", fontsize = 20)
        # plot w
        plt.figure(1)
        for i in range(0, len(ws)):
```

```
            plt.plot(ws[i])
        plt.ylim(0, 4.5)
        plt.xlim(-10, measures)
        plt.xticks(np.arange(0, measures + 1, 50), fontsize = 20)
        plt.yticks(fontsize = 20)
        plt.legend(['lambda value 0.001', 'lambda value 0.01', 'lambda value 0.1', 'lambda value 1'],
loc='upper left', fontsize = 20)
        plt.xlabel("Steps / 30", fontsize = 20)
        plt.ylabel("w at every 30 steps", fontsize = 20)
        plt.show()
    def _objective(self, margins):
        return 0.5 * self.a.dot(self.a) + np.maximum(0, 1 - margins).sum()
    def linear_function(self, X):
        return X.dot(self.a) + self.b
    def predict(self, X):
        return np.where(self.linear_function(X) >= 0, 1, -1)
    def score(self, X, Y):
        P = self.predict(X)
        return np.mean(Y == P)
    def fit(self, X, Y, Xval, Yval, lambdaVals, m, n):
        N, D = X.shape
        self.N = N
        self.evaluationStep = 30
        lambdaVals = [1e-3, 1e-2, 1e-1, 1]
        accuracies = []
        ws = []
        valuation = []
        for lambdaVal in lambdaVals:
            self.a = np.zeros(D)
            self.b = 0
            accuracy = []
            w = []
            for season in range(self.Seasons):
                season += 1
                lr = 1 / ((m * season) + n)
                # Take out 50 examples for testing at every 30 step
                combinedXY = np.concatenate((X, np.matrix(Y).T), axis=1)
                accuracyData, trainData = splitDataseByNumber(combinedXY, 50)
                accuracyX = accuracyData[:, :-1]
                accuracyY = np.array(accuracyData[:, -1]).flatten()
                trainDataX = trainData[:, :-1]
                trainDataY = np.array(trainData[:, -1]).flatten()
                # gradient descent
                for step in range(self.Steps):
                    if (step % self.evaluationStep == 0):
                        accuracy.append(self.score(accuracyX, accuracyY))
                        w.append(self.a.dot(self.a))
                    k = np.random.choice(N - 50, 1)[0]
                    Xk = np.array(trainDataX[k, :]).flatten()
                    yk = trainDataY[k]
                    margins = yk * self.linear_function(Xk)
                    if (margins >= 1):
                        pa = lambdaVal * self.a
                        pb = 0
                    else:
                        pa = lambdaVal * self.a - yk * Xk
                        pb = -yk
                    self.a -= lr * pa
                    self.b -= lr * pb
            accuracies.append(accuracy)
            ws.append(w)
            # Calculate accuracy of the valuation dataset
            valuation.append(self.score(Xval, Yval))
        # Get the lambda value that achieve the maximum accuracy of the valuation dataset
        for i in range(0, len(valuation)):
            print("Accuracy of the valuation dataset is {} with lambda value {} and m value
{}".format(valuation[i], lambdaVals[i], m))
        indBest = valuation.index(max(valuation))
```

```
        print("The best value of lambda is {} with the maximum accuracy {} with m value
{}".format(lambdaVals[indBest], valuation[indBest], m))
        # Plot accuracy
        # self.plotResult(accuracies, ws)
        return [lambdaVals[indBest], valuation[indBest]]
    def testSetScore(self, lambdaVal, trainDataset, testDataset, m, n):
        # Select the columns with continuous values
        trainX = trainDataset[:, [0, 2, 4, 10, 11, 12]].astype(np.float)
        testX = testDataset[:, [0, 2, 4, 10, 11, 12]].astype(np.float)
        # Normalization of the dataset features
        trainDataX = normalization(trainX)
        testX = normalization(testX)
        # Update class values
        trainDataY = np.where(trainDataset[:, -1] == ' <=50K', -1, 1)
        N, D = trainDataX.shape
        self.a = np.zeros(D)
        self.b = 0
        for season in range(self.Seasons):
            season += 1
            lr = 1 / ((m * season) + n)
            # gradient descent
            for step in range(self.Steps):
                k = np.random.choice(N - 50, 1)[0]
                Xk = np.array(trainDataX[k, :]).flatten()
                yk = trainDataY[k]
                margins = yk * self.linear_function(Xk)
                if (margins >= 1):
                    pa = lambdaVal * self.a
                    pb = 0
                else:
                    pa = lambdaVal * self.a - yk * Xk
                    pb = -yk
                self.a -= lr * pa
                self.b -= lr * pb
        # Predict the test dataset
        predictY = np.where(self.linear_function(testX) >= 0, ">50K", "<=50K")
        np.savetxt('submission.txt', predictY, fmt='%s')
def main():
    trainFilename = 'train.txt'
    testFilename = 'test.txt'
    trainDataset = loadCsv(trainFilename)
    testDataset = loadCsv(testFilename)
    splitRatio = 0.9
    accuracy = 0
    # Split and rearrange the dataset
    Xtrain, Ytrain, Xval, Yval = arrangeDataset(trainDataset, splitRatio)
    model = SVM(50, 300)
    lambdaVals = [1e-3, 1e-2, 1e-1, 1]
    model.fit(Xtrain, Ytrain, Xval, Yval, lambdaVals, 0, 50)
    model.fit(Xtrain, Ytrain, Xval, Yval, lambdaVals, 0.01, 50)
    # print("m is {}, n is {}".format(m, n))
    # predict test dataset
    model.testSetScore(0.001, trainDataset, testDataset, 0.01, 50)
main()
```