CS498 AMO Homework 2

Team :
Minyuan Gu (minyuan3@illinois.edu, netid minyuan3)
Yanislav Shterev (shterev2@illinois.edu, netid shterev2)

Page 1 (15 points)

Leaderboard

Search

RANK	SUBMISSION NAME	ACCURACY
2	shterev.yan	83.19
2	MINYUAN GU	83.19

STUDENT
Minyuan Gu

AUTOGRADER SCORE
83.19 / 100.0

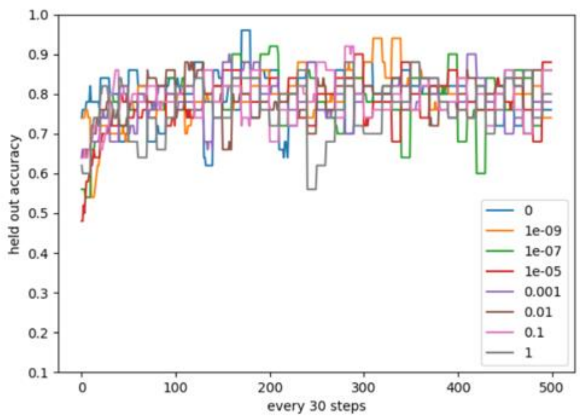
STUDENT
Yanislav Shterev

AUTOGRADER SCORE
83.19 / 100.0

Page 2 (20 points)

A plot of the validation accuracy every 30 steps, for each value of the regularization constant.

Please refer to the figure 1 (left side) for the plot of every 30 steps on the held out 50 samples, required by the homework. Figure 2 is done on a much large validation to see the smooth converge (for comparison purpose), which we will discuss below.



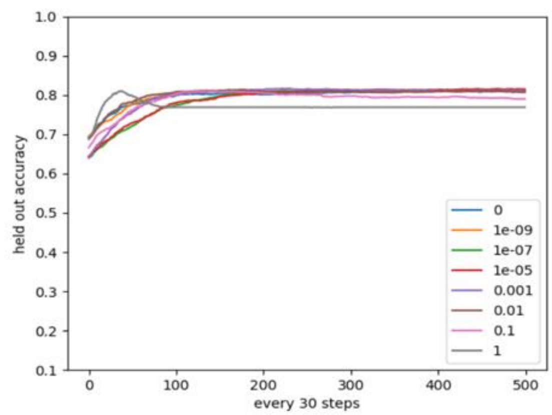


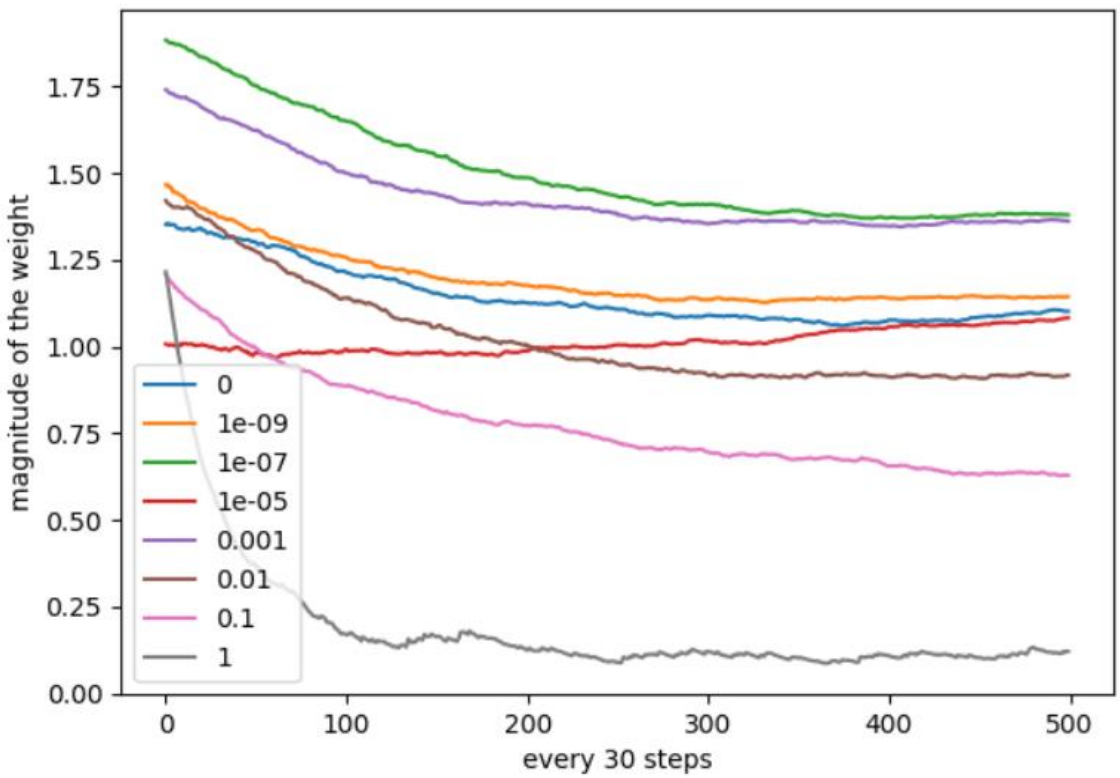
Figure 1 Accuracy on 50 Held out sample, required by the HW

Figure 2 Accuracy on larger set (10% validation for comparison only)

Due to the facts that accuracy was calculated on 50 samples held out set (required by HW), this introduced high variance in the accuracy. We tried to increase the number of samples in the held out validation set, it will smooth out the variance, see Figure2 as example. However, from the first chart (Figure 1), we can still see the trend of increasing accuracy oscillating along with the training seasons.

Page 3 (20 points)

A plot of the magnitude of the coefficient vector every 30 steps, for each value of the regularization constant.



Page 4 (25 points)

The lambda consideration

The best value of the regularization rate was while lambda is 0.01 (1e-2) based on the validation set accuracy we had:

```
Test data accuracy is for lambda  0  is:  79.24914675767918 %
Test data accuracy is for lambda  1e-09  is:  79.226393629124 %
Test data accuracy is for lambda  1e-07  is:  79.24914675767918 %
Test data accuracy is for lambda  1e-05  is:  79.5449374288964 7 %
Test data accuracy is for lambda  0.001  is:  78.86234357224117 %
Test data accuracy is for lambda  0.01  is:  80.13651877133105 %
Test data accuracy is for lambda  0.1  is:  79.29465301478953 %
Test data accuracy is for lambda  1  is:  76.01820250284415 %
```

Actually all have very similar performance except 1. We decided to choose 1e-2 because it has best accuracy; also slightly larger regularization constant (comparing to 1e-5 and below) assists in preventing overfitting and reduces the chance of high variance which does not generalize well for future data. We also tested the extreme cases, for example lambda=0 (disabled regularization) and lambda=1 (lean more to regularization); we observed both cases are far from ideal.
By increasing lambda, the penalization factor increased on the weight magnitude, while features contribution to the learning was decreased. It could start causing model under-fitting if too large. This is shown above why lambda =1 has the lowest accuracy. On the other side, having low value of lambda caused the model over-fitting and resulted in lower accuracy on validation set.

The learning rate (step length) consideration.

The corresponding m and n parameters are part of the learning rate formula in our code: lr = step_length_m/(20*i+step_length_n)

where we have step_length_m = 1 and step_length_n = 1000, i is the current season. For the 20 in from of i (**20***i+….), 20 is chosen as the multiplying factor to trim down the learning rate at later seasons (we also tried setting this number to 0.1, 1, 2, 10, they have different LR decreasing speeds but it didn't show much difference in the final accuracy if initial rate was set properly).

From above, it means we will have a starting learning rate of 0.001 (1/1000) and slowly trimming down to 0.0005(1/20*50+1000) if total season is 50. We also tried other step_length_n, e.g. 10, 50, 100, 200, and 2000; it proved they all converged at different speeds.

If learning rate is too small, we will require more seasons to train the model until it is stable; if the learning rate is too large, it will become hard to converge at later stage since larger value causes the oscillating of the weights (e.g. making occasional large, bad moves). We chose 0.001 based on the facts from above graph: it converged reasonably fast and stable, and small learning rate can allow it to converge better at later seasons. Especially in this case of batch size =1 (Stochastic Gradient Descent), smaller learning rate are recommended due to the fact that gradient is calculated on only one sample which means more variance. We don't want just a 'bad' sample to impact our model.

Another factor is that initial weights also played a role on the final accuracy and learning rate. We used randomly initialized weight and choose small learning rate carefully in case of initial bad weights. Having that said, we also noticed if we put a stop of the training once we reached a certain accuracy (e.g. record high) on the validation set, with batch size = 3 or 5, it seems initial learning rate of 0.2 (step_length_n = 5) can still perform well in the test set on auto-grader; but this is out of scope of discussion of this home work.

Page 5 A screenshot of your code.

Training of an SVM, including but not limited to SGD.

```
class supportVectorMachine:
    def __init__(self, weight, b=0.0, reg_lambda=1e-1, step_length=0.1):
        self.reg_lambda = reg_lambda
        self.weight = np.array(weight)
        self.b = b
        self.X = np.array([])
        self.Y = np.array([])
        self.cost = 0
        self.training_cost = np.array([])
        self.step_length = step_length

    def cost_function(self, X, Y):
        self.training_cost = 1 - (np.dot(X, self.weight)+self.b)*Y
        self.training_cost[self.training_cost<0] = 0
        self.cost = np.mean(self.training_cost) + self.reg_lambda*np.dot(self.weight.T, self.weight)/2
        print(" training cost is ", self.cost)

    def update_weight(self):
        weight_to_update=self.X*self.Y.reshape((self.Y.shape[0],1))
        zero_cost_matrix = 1 - (np.dot(self.X, self.weight)+self.b)*self.Y
        zero_cost_matrix[zero_cost_matrix<0]=0
        weight_to_update = weight_to_update*((zero_cost_matrix!=0).reshape((zero_cost_matrix.shape[0],1)))
        weight_to_update = -(1/self.X.shape[0])*np.sum(weight_to_update,axis=0)+self.reg_lambda*self.weight
        self.weight = self.weight - self.step_length*weight_to_update
        # to update b
        self.b = self.b - self.step_length*(-np.dot(self.Y, 1*(zero_cost_matrix!=0))/self.X.shape[0])

    def StochasticGradientDesc(self, X, Y):
        self.X = X
        self.Y = Y
        self.update_weight()

    def predict(self, X):
        return 2*((np.dot(X, self.weight)+self.b)>0)-1

...(we skip a few lines of codes eg. Definition of hyper-parms and data loading, please refer to the full code)
weight = np.random.rand(X.shape[1])
svm = supportVectorMachine(weight=weight, reg_lambda=regularisation_lambda[idx_lambda])
for i in range(total_season):
    print("******season: ", i," *******")
    lr = step_length_m/(20*i+step_length_n)
    svm.set_learningRate(lr)

    np.random.shuffle(train_data)
    held_out = train_data[:50, :]
    train = train_data[50:, :]
    for j in range(1, steps+1):
        selected = np.random.randint(train.shape[0], size=batch_size)
        svm.StochasticGradientDesc(train[selected, :-1], train[selected, -1])
        if j % 30 == 0:
            print("--->Step: ", j, " <----")
            validation_result = svm.predict(held_out[:, :-1])
            validation_accuracy = sum(validation_result == held_out[:, -1]) / held_out.shape[0]
            accuracy_history[idx_lambda, int((i*steps+j)/30)-1] = validation_accuracy
            weight_magnitude_history[idx_lambda, int((i*steps+j)/30)-1] = math.sqrt(np.sum(svm.weight ** 2))
```

Testing of an SVM.

1. The following is used to test SVM over 10% validation set:

```
test_result = svm.predict(hyperParmSearch_data[:, :-1])
test_accuracy = sum(test_result == hyperParmSearch_data[:, -1]) / hyperParmSearch_data.shape[0]
```

2. The following is used to test SVM over the test set (for grading on auto grader)

```
grader_data = []
with open('./homework2/test.txt', newline='') as f:
    reader = csv.reader(f, delimiter=',')
    for row in reader:
        grader_data.append(row)

grader_data = np.array(grader_data)
grader_X = grader_data[:, (0,2,4,10,11,12)].astype(float)
# rescale the features to same variance and zero means.
grader_X = (grader_X - np.mean(grader_X, axis=0))/np.std(grader_X,axis=0)
grader_result = svm.predict(grader_X)
save_for_submission(grader_result)
```

Page 6+ Full codes

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


class supportVectorMachine:
    def __init__(self, weight, b=0.0, reg_lambda=1e-1, step_length=0.1):
        self.reg_lambda = reg_lambda
        self.weight = np.array(weight)
        self.b = b
        self.X = np.array([])
        self.Y = np.array([])
        self.cost = 0
        self.training_cost = np.array([])
        self.step_length = step_length

    def cost_function(self, X, Y):
        self.training_cost = 1 - (np.dot(X, self.weight)+self.b)*Y
        self.training_cost[self.training_cost<0] = 0
        self.cost = np.mean(self.training_cost) + self.reg_lambda*np.dot(self.weight.T, self.weight)/2
        print(" training cost is ", self.cost)

    def update_weight(self):
        weight_to_update=self.X*self.Y.reshape((self.Y.shape[0],1))
        zero_cost_matrix = 1 - (np.dot(self.X, self.weight)+self.b)*self.Y
        zero_cost_matrix[zero_cost_matrix<0]=0
        weight_to_update = weight_to_update*((zero_cost_matrix!=0).reshape((zero_cost_matrix.shape[0],1)))
        weight_to_update = -(1/self.X.shape[0])*np.sum(weight_to_update,axis=0)+self.reg_lambda*self.weight
        self.weight = self.weight - self.step_length*weight_to_update
        # to update b
        self.b = self.b - self.step_length*(-np.dot(self.Y, 1*(zero_cost_matrix!=0))/self.X.shape[0])

    def StochasticGradientDesc(self, X, Y):
        self.X = X
        self.Y = Y
        self.update_weight()

    def predict(self, X):
        return 2*((np.dot(X, self.weight)+self.b)>0)-1

    def set_learningRate(self, lr):
        self.step_length = lr


def save_for_submission(results):
    fobj = open('./homework2/submission.txt', 'a+')
    for i in results:
        if i >= 1:
            fobj.write('>50K\n')
        else:
            fobj.write('<=50K\n')
    fobj.close()


#################################################################
#     Import training data, shuffle, rescale & split            #
#################################################################
data = []
X = []
Y = []
# import the data from the csv.
with open('./homework2/train.txt', newline='') as f:
    reader = csv.reader(f, delimiter=',')
    for row in reader:
        data.append(row)

data = np.array(data)
np.random.shuffle(data)
# extract only continuous variable values to form X
X = data[:, (0,2,4,10,11,12)].astype(float)
# extract last col to form classes of 1 for >50K and -1 for <=50K
Y = 2*(data[:, 14] == ' >50K')-1
# rescale the features to same variance and zero means.
X = (X - np.mean(X, axis=0))/np.std(X,axis=0)
rescaled_data = np.column_stack((X,Y))

split_idx = int(data.shape[0]*0.1)
```

```python
hyperParmSearch_data = rescaled_data[:split_idx, :]
train_data = rescaled_data[split_idx:, :]

#################################################################
#                    Hyper Parameters definition                #
#################################################################
regularisation_lambda = [0,1e-9, 1e-7, 1e-5, 1e-3, 1e-2, 1e-1, 1]
step_length_m = 1
step_length_n = 1000
total_season = 50
steps = 300
batch_size = 1

#################################################################
#             Training using different lambda values            #
#################################################################
# following history record the held out accuracy every 30 steps.
accuracy_history = np.zeros((len(regularisation_lambda), int(steps*total_season/30)))
weight_magnitude_history = np.zeros((len(regularisation_lambda), int(steps*total_season/30)))
# following accuracy report each lambda's performance against validation set.
final_validation_accuracy_history = np.zeros(len(regularisation_lambda))
svm = None
max_achieved_accuracy = 0
max_achieved_weight = []
for idx_lambda in range(len(regularisation_lambda)):
    weight = np.random.rand(X.shape[1])
    svm = supportVectorMachine(weight=weight, reg_lambda=regularisation_lambda[idx_lambda])
    for i in range(total_season):
        print("******season: ", i," *******")
        lr = step_length_m/(20*i+step_length_n)
        svm.set_learningRate(lr)

        np.random.shuffle(train_data)
        held_out = train_data[:50, :]
        train = train_data[50:, :]
        for j in range(1, steps+1):
            selected = np.random.randint(train.shape[0], size=batch_size)
            svm.StochasticGradientDesc(train[selected, :-1], train[selected, -1])
            if j % 30 == 0:
                print("--->Step: ", j, " <----")
                validation_result = svm.predict(held_out[:, :-1])
                validation_accuracy = sum(validation_result == held_out[:, -1]) / held_out.shape[0]
                accuracy_history[idx_lambda, int((i*steps+j)/30)-1] = validation_accuracy
                weight_magnitude_history[idx_lambda, int((i*steps+j)/30)-1] = math.sqrt(np.sum(svm.weight **
2))

    test_result = svm.predict(hyperParmSearch_data[:, :-1])
    test_accuracy = sum(test_result == hyperParmSearch_data[:, -1]) / hyperParmSearch_data.shape[0]
    final_validation_accuracy_history[idx_lambda] = test_accuracy
    if test_accuracy >= max_achieved_accuracy:
        max_achieved_accuracy = test_accuracy
        max_achieved_weight = (svm.weight, svm.b, idx_lambda)

#################################################################
#            Plot the graph for different lambdas               #
#################################################################
x_axis = range(int(steps*total_season/30))
plt.subplot(1, 2, 1)
for idx_lambda in range(len(regularisation_lambda)):
    print(" Test data accuracy is for lambda ",regularisation_lambda[idx_lambda]," is: ",
final_validation_accuracy_history[idx_lambda]*100, "%")
    plt.plot(x_axis, accuracy_history[idx_lambda])
plt.ylim((0.1,1))
plt.legend(regularisation_lambda, loc='lower right')
plt.xlabel('every 30 steps')
plt.ylabel('held out accuracy')
plt.subplot(1, 2, 2)
for idx_lambda in range(len(regularisation_lambda)):
    plt.plot(x_axis, weight_magnitude_history[idx_lambda])
plt.xlabel('every 30 steps')
plt.ylabel('magnitude of the weight')
plt.legend(regularisation_lambda, loc='lower left')
plt.show()


#################################################################
#          Predict on the test set for submission               #
#################################################################
svm.weight = max_achieved_weight[0]
svm.b = max_achieved_weight[1]
grader_data = []
```

```
with open('./homework2/test.txt', newline='') as f:
    reader = csv.reader(f, delimiter=',')
    for row in reader:
        grader_data.append(row)

grader_data = np.array(grader_data)
grader_X = grader_data[:, (0,2,4,10,11,12)].astype(float)
# rescale the features to same variance and zero means.
grader_X = (grader_X - np.mean(grader_X, axis=0))/np.std(grader_X,axis=0)
grader_result = svm.predict(grader_X)
save_for_submission(grader_result)
```

Libraries used & Reference:

David Forsyth's book - Probability and Statistics for Computer Science
David Forsyth's book - Applied Machine Learning
Trevor Walker's lecture and sample code – CS-498 Lecture videos
csv – for reading data from csv format: https://docs.python.org/3/library/csv.html
Adult dataset - training dataset
https://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/train.txt
Testing dataset https://courses.engr.illinois.edu/cs498aml/sp2019/homeworks/test.txt
Numpy - http://www.numpy.org/
matplotlib - to plot the accuracy and magnitude: https://matplotlib.org/