CSE 250C HW3

Yansong Gao A53100210, Jinyuan Li A53101535

June 2016

1 Introduction

The goal of this project is to explore the insights about the learnability of Convex-Lipschitz-Bounded problems by conducting numerous experiments with respect to different problem configurations. To be more specific, we want to experimentally prove the correctness of the learnability theory of Convex-Lipschitz-Bounded problem as well as explore the impacts of different parameter configurations on the learnability of the learner. In this project, we adopted Logistic function as the convex loss function and Stochastic Gradient Descent (SGD) algorithm as the learner. The followings are the goals that we wanted to achieve in this project.

- Evaluate the impact of different numbers of input examples on the learning performance of SGD learner (aka. the learnability of the Hypothesis class).
- Evaluate the impact of different data space and parameter set shape on the performance of SGD learner (aka. the learnability of the Hypothesis class).
- Evaluate the impact of different underlying distribution of examples on the performance of SGD learner (aka. the learnability of the Hypothesis class).

To measure the performance of SGD learner, we use both **expected risk** and **expected true error** as two metrics. They are actually corelated because a bigger expected risk indicates a bigger expected true error, vise versa. So we conducted multiple experiments with different settings and measure these two metrics. To study the behavior of the learner with respect to single variable, we conducted experiments which just changed this single variable and kept other variables remain unchanged. We then evaluate the affects by plotting the performance curves into plots. With these experiments, we expected to know the

2 Symbol Index Listing

For the code part, there are three major functions:

- trainLogRegres function
- testLogRegres function
- generate$_e$*xamplepart*

And for trainLogRegres function, there are four input parameters.

- train_x is a matrix datatype, each row stands for one sample, and each column stands for one feature.
- train_y is also a matrix datatype, each row is the corresponding label of the sample.
- alpha is the step size, which we set to be 0.01. We determined the best value of α by trials.
- maxIter is the number of iteration.

And the return value of this function is weights, it is the calculated weights for each feature.

For testLogRegres function, there are three input parameter.

- weights is the weights calculated by trainLogRegres function.
- test_x is a matrix datatype, each row stands for one test data, each column stands for one feature.
- test_y is also a matrix datatype, each row is the corresponding label of the sample.

The return value is the accuracy of the weights.

For generate$_e$$xamplepart, weusedalltheuserdefinedparametersfor\sigma$ and μ. Then we also calculate the norm of data points for projection.

3 Experiment Design

3.1 Generate Examples

The general idea of generating examples from the underlying distribution is to use randomized sampling method to generate four floating numbers for each example because the dimension of each $\boldsymbol{x}$ is 4. According to the homework specifications, the distribution of each example depends on its corresponding label. Thus, we first generate the label to be either -1 or +1 for each example. Then if the label is -1, we use the method illustrated above to generate 4 independent numbers as one example from a Gaussian distribution $\mathcal{N}(-\frac{1}{4}, \sigma^2)$ where σ has two option values being (0.05, 0.25). Then after generating raw numbers from the Gaussian distribution, we have to take one more step to project the data that lies outside the boundary of the data domain onto the surface of the domain. This step will make sure that all the examples are within the boundary of the domain.

3.2 Learning

The learning process for each setting (ie. different training set size, different data generating distribution and different data domain shape) is the same. First, we adopt the online learning approach which feed the generated training examples one by one to the learner each iteration. The SGD learner will calculate the stochastic gradient based on the current weight vector and the example vector as well as the example label. Then weight vector is updated by multiplying with the learning rate of each iteration. After feeding all the training examples, the output weight vector is the average of all the weight vectors learned in between.

Since we also want to get the parameter ρ for the Lipschitzness of the loss function and the parameter M for the boundary of the convex set C, we also keep track on the gradients of the loss function and the norm of weight vectors during the training process.

$$\forall \boldsymbol{w} \in \mathbb{R}^d \ ||\nabla f(\boldsymbol{w})|| \leq \rho \tag{1}$$

$$\exists M > 0 s.t. \forall \boldsymbol{w} \in C, ||\boldsymbol{w}|| \leq M \tag{2}$$

3.3 Changing variables

Under each scenario, we focus on studying the impact of two variables on the performance of SGD learner, ie. the number of training examples and the standard deviation σ of the underlying Gaussian distribution for generating data points.

In order to evaluate the impact of input example numbers on the performance of SGD learner, we generate different number of examples for each experiment but keep all the other variables unchanged. We choose the numbers n to be 50, 100, 500 and 1000 and train our learner with fresh generated examples. From the learnability theorem, we have enough reason to expect that both of the expected risk and expected true error to be smaller as we feed in more examples.

In order to evaluate the impact of different underlying data generating distribution on the performance of our learner, we choose σ to be either (0.05, 0.25) as the standard deviation of the Gaussian distribution. So for each value we chose, we conduct the same set of experiments with different number of input examples and measure the results for evaluation. The detail of evaluating is discussed in the next subsection.

3.4 Evaluating (Testing)

After feeding all of the training data to the SGD learner, we obtain an optimized weight vector for testing. We uniformly use 400 fresh examples as the test set. To visually observe the performance differences of our learner trained under different parameter settings, we measure both of the expected risk and the expected true error incurred by the leaner against each test example. Then we plot all expected risks into one plot and all expected true error into another plot. By comparing the curves, we can easily discover the influence of these variables on the performance of the learner. The plots will be shown in section **??**.

4 Results and Evaluation

We conducted all the experiments and listed all the measurements below. There are totally 4 tables showing the expected true error and the expected risk of our SGD learner under different settings. The four corresponding plots are also displayed below.

The SGD performance (both expected true error and expected risk) under "hyperball" scenario is better than the performance under "hypercube" scenario. This is reasonable because the M-bound of our convex parameter set C is smaller for "hyperball" scenario. Therefore, according to the learnability theorem, we need fewer training samples to learn the hypothesis class if we have a smaller M with the same ρ. Our measurement also proves this argument.

As for σ, we see that the influence of σ is more significant on the expected true error than the expected risk. In general, a bigger σ, which means a more emission distribution of examples, will lead to higher expected true error and risk. This is reasonable because it's of course easier to learn in a more centralized data space than a huge emission data space.

Also, we can see clearly from the figures that the performance of our learner will improve as the training example set grows bigger. This again proves the learnability theorem that with more training data, we can achieve lower true error with the learner.

	n=50	n=100	n=500	n=1000
sigma = 0.05	0.012625	0.0	0.0	0.0
sigma = 0.25	0.144250	0.077500	0.012500	0.012

Table 1: expected true error of hyperball

	n=50	n=100	n=500	n=1000
sigma = 0.05	0.677565	0.66280	0.56215	0.46914
sigma = 0.25	0.67742	0.66326	0.56629	0.484003

Table 2: expected risk of hyperball

	n=50	n=100	n=500	n=1000
sigma = 0.05	0.075499	0.017250	0.0	0.0
sigma = 0.25	0.15725	0.07199	0.028875	0.024625

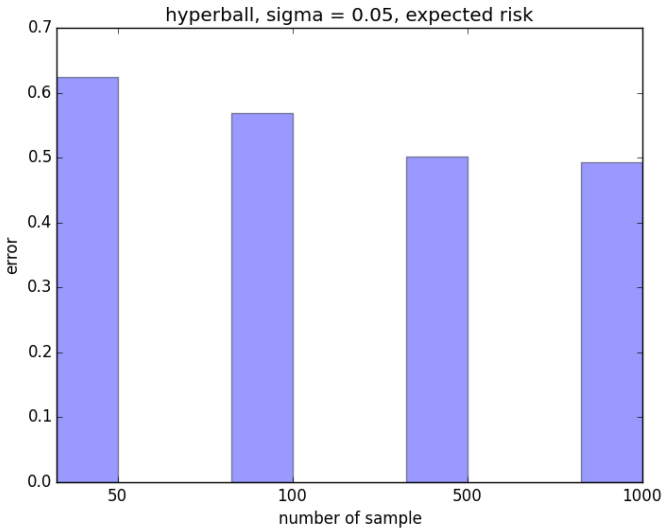
Table 3: expected true error of hypercube

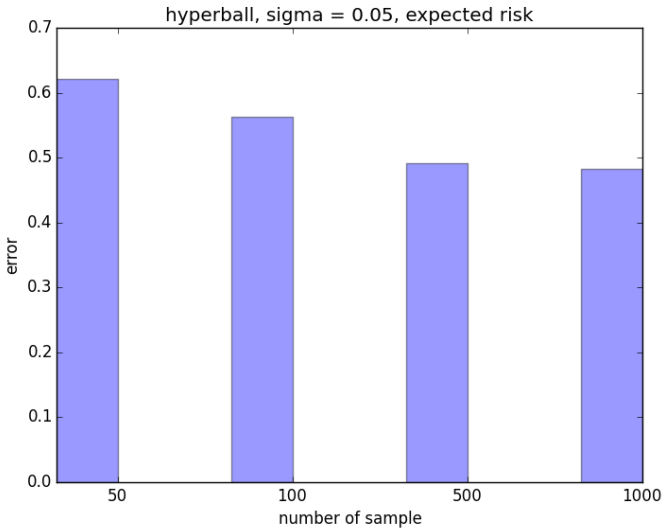
5 Conclusion

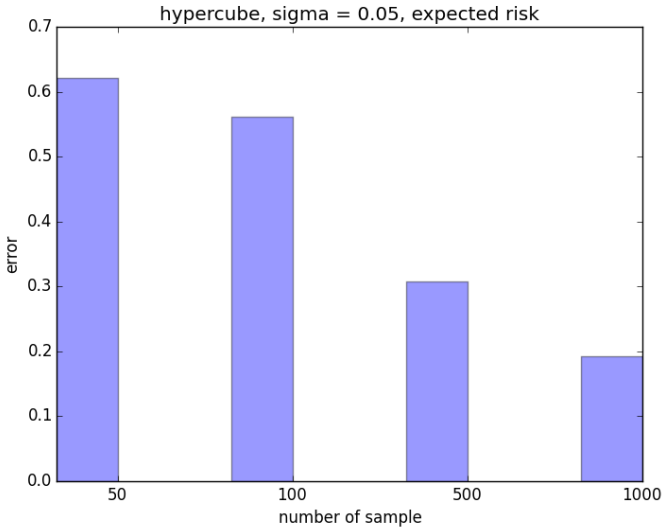
In the project, we proved the learnability theorem for Convex-Lipschitz-Bounded problems. We implemented a Stochastic Gradient Descent learner and conducted multiple experiments with various settings. In conclusion, we proved that bigger training set can always result in a better true error and risk. We also proved that it's easier to learn or the performance of the learner will be better given the same training set if the

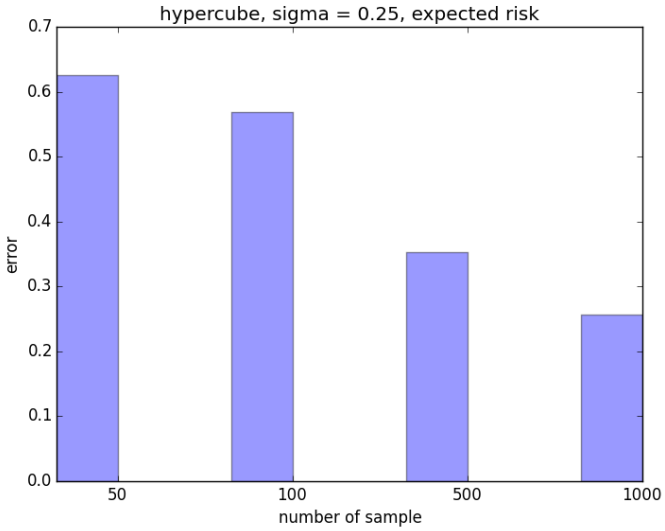
	n=50	n=100	n=500	n=1000
sigma = 0.05	0.677548	0.66354	0.561734	0.516686
sigma = 0.25	0.67834	0.664614	0.569393	0.526066

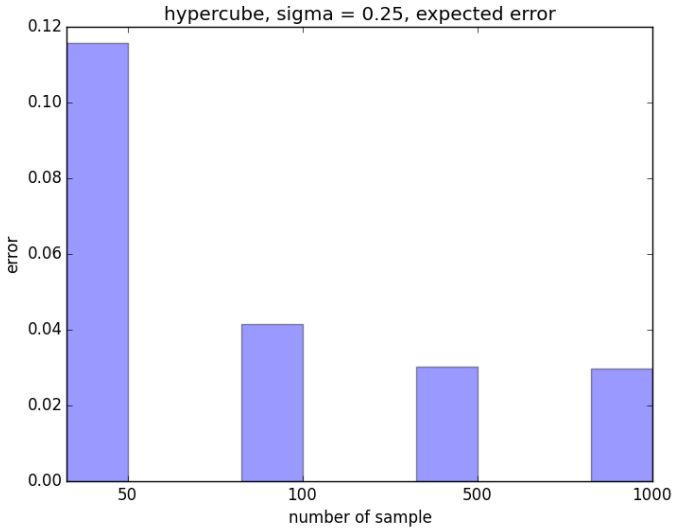
Table 4: expected risk of hypercube

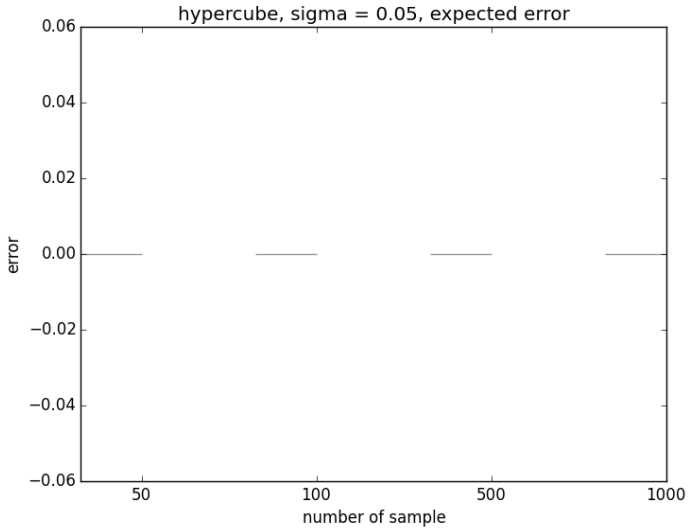


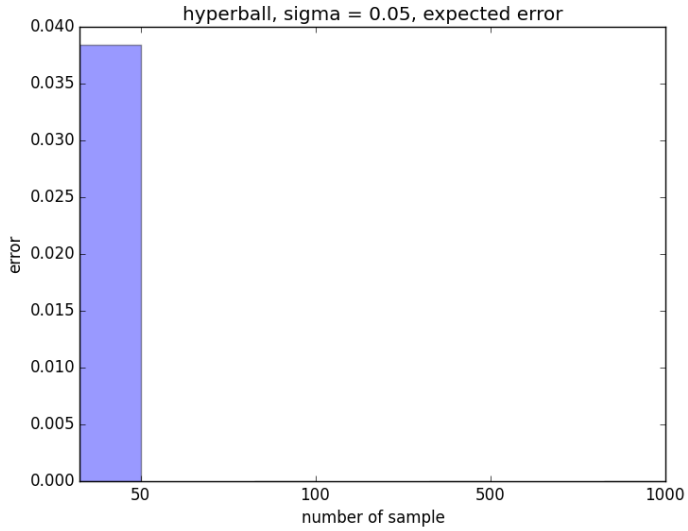


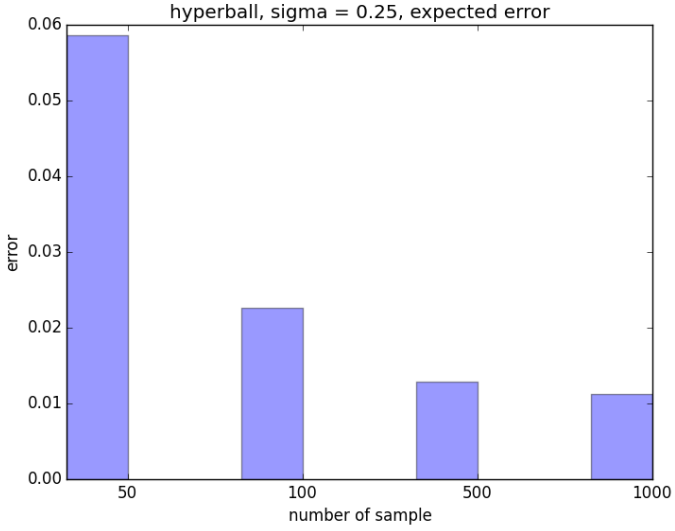

hypercube, sigma = 0.05, expected risk
error
number of sample
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
50
100
500
1000


hypercube, sigma = 0.25, expected risk
error
number of sample
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
50
100
500
1000


hypercube, sigma = 0.25, expected error
0.12
0.10
0.08
0.06
0.04
0.02
0.00
error
50
100
500
1000
number of sample


hypercube, sigma = 0.05, expected error
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
error
50
100
500
1000
number of sample


hyperball, sigma = 0.05, expected error
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
error
50
100
500
1000
number of sample


hyperball, sigma = 0.25, expected error
0.06
0.05
0.04
0.03
0.02
0.01
0.00
error
50
100
500
1000
number of sample

parameter set C has a tighter bound M. Finnaly, we proved that if the underlying distribution for generating examples is more centralized, then the performance of the SGD learner will be better.

One more thing is that we fixed the learning rate for each iteration. For future work, we will use a dynamic learning rate setting to improve the performance of learning.

6 Contribution

The team worked together on coding and writing report. Jingyuan Li focused on the data-generating part, plotting part and report. Yansong Gao focused on implementing the core functions of SGD learner.

7 Appendix

```
from numpy import *
import matplotlib.pyplot as plt
import time
import generate_sample as gs

MEAN = 0.25
MAX_ITR = 1
ALPHA = 0.01
TESTSET_SIZE = 400
SCENARIO = 'hyperball'
# calculate the sigmoid function
def sigmoid(inX):
        return 1.0 / (1 + exp(-inX))

def loss_gradient(w, x, y):

    return -y * exp(-y * w.dot(x)) * x / (1 + exp(-y * w.dot(x)))


def loss(weights, inX, inY):
        return log(1+exp(-inX * inX * weights))


# train a logistic regression model using SGD
# input: train_x is a mat datatype, each row stands for one sample
#                train_y is mat datatype too, each row is the corresponding label
#                alpha is the step size
#         maxIter is the number of iterations
def trainLogRegres(train_x, train_y, alpha, maxIter, scenario):
        # calculate training time
        startTime = time.time()
        numSamples, numFeatures = shape(train_x)
        weights = zeros((numSamples +1, 5))
        for t in range(numSamples):
                G = loss_gradient(weights[t], train_x[t], train_y[t])
                weights[t + 1] = gs.projection(scenario ,weights[t] - alpha * G)
        return mean(weights[1:], axis = 0)

# test your trained Logistic Regression model given test set
```

```
def testLogRegres(weights, test_x, test_y):
        numSamples, numFeatures = shape(test_x)
        matchCount = 0
        errorCount = 0
        totalLoss = 0
        for i in xrange(numSamples):
                true_result = False
                if test_y[i] == -1:
                        true_result = False
                else:
                        true_result = True
                predict = sigmoid(inner(test_x[i] , weights)) > 0.5

                if predict == true_result:
                        matchCount += 1
                else:
                        errorCount += 1
        accuracy = float(matchCount) / numSamples
        errorRate = float(errorCount) / numSamples
        totalLoss = gs.loss(weights, test_y, test_x)

        return errorRate, float(sum(totalLoss)) / numSamples


# show your trained logistic regression model only available with 2-D data
def showLogRegres(weights, train_x, train_y):
        # notice: train_x and train_y is mat datatype
        numSamples, numFeatures = shape(train_x)
        if numFeatures != 3:
                print "Sorry! I can not draw because the dimension of your data is not 2!"
                return 1

        # draw all samples
        for i in xrange(numSamples):
                if int(train_y[i, 0]) == 0:
                        plt.plot(train_x[i, 1], train_x[i, 2], 'or')
                elif int(train_y[i, 0]) == 1:
                        plt.plot(train_x[i, 1], train_x[i, 2], 'ob')

        # draw the classify line
        min_x = min(train_x[:, 1])[0, 0]
        max_x = max(train_x[:, 1])[0, 0]
        weights = weights.getA()  # convert mat to array
        y_min_x = float(-weights[0] - weights[1] * min_x) / weights[2]
        y_max_x = float(-weights[0] - weights[1] * max_x) / weights[2]
        plt.plot([min_x, max_x], [y_min_x, y_max_x], '-g')
        plt.xlabel('X1'); plt.ylabel('X2')
        plt.show()


def main():
        n = [50, 100, 500, 1000]
        sigma = [0.05, 0.25]
        scenario = ['hypercube', 'hyperball']
```

```
        for sub_s in scenario:
                for sub_sigma in sigma:
                        expected_err = []
                        expected_risk = []
                        test_x, test_y = gs.generate_samples(TESTSET_SIZE, MEAN, sub_sigma
                        for sub_n in n:
                                true_err_exp = []
                                risk_exp = []
                                for i in range(20):
                                        ## generate training set
                                        train_x, train_y = gs.generate_samples(sub_n, MEAN
                                        learned_weights = trainLogRegres(train_x, train_y,

                                        true_err, risk = testLogRegres(learned_weights, tes
                                        true_err_exp.append(true_err)
                                        risk_exp.append(risk)
                                expected_err.append(sum(true_err_exp)/20)
                                expected_risk.append(sum(risk_exp)/20)
                        print "scenario: ", sub_s, "sigma = ", sub_sigma, "expected true e
                        ## plot two figures here

if __name__ == '__main__':
        main()

__author__ = 'Jingyuan Li'

import numpy as np

''' input: desired dataset size, mean and std of Gaussian distribution, and the scenario
    output: the generated dataset with specific size from Gaussian, as well as its correps
'''

def generate_samples(sample_size, mean, std, scenario):
        y = np.sign(np.random.uniform(-0.25, 0.25, sample_size))
        x = []
        for label in y:
                x.append(np.append(np.random.normal(mean * label, std, 4), 1.0))

        return projection_inner(scenario, x), y

''' input: scenario and dataset
    output: the dataset after projection
'''
def projection_inner(scenario, x):
        if scenario is 'hypercube':
                for samples in x:
                        for dim in samples:
                                if dim > 1: dim = 1
                                elif dim < -1: dim = -1
        elif scenario is 'hyperball':
                for samples in x:
                        norm = np.linalg.norm(samples)
                        if norm > 1:
                                samples = np.divide(samples , np.ones((1,5))*(norm))
```

```
        return x

''' input: scenario and example vector
    output: the vector after projection
    This function is the same as above but with different input
'''
def projection(scenario, vector):
        if scenario is 'hypercube':
                for dim in vector:
                        if dim > 1: dim = 1
                        elif dim < -1: dim = -1
        elif scenario is 'hyperball':
                norm = np.linalg.norm(vector)
                if norm > 1:
                        vector = np.divide(vector , np.ones((1,5))*(norm))
        return vector

## calculate the loss
def loss(weights, y, x):
        loss = []
        for i in range(y.size):
                loss.append(np.log(np.exp(np.inner(weights, x[i]) * (-y[i])) + 1))
        return loss
```