Project: Stochastic Gradient Descent

Authors: Hangxiao Zhu, Adam Wang

1 Introduction:

The overall goals of this project is as following:

1. Implement the SGD algorithm.
2. Apply the implemented SGD algorithm to the generated dataset and try to find the weights that minimize the true risk.
3. Perform the experiment by using different hyperparameters (variance, training size) and observe the difference in result in terms of logistic loss and classification error.

Here is the outline of the SGD algorithm:

Inputs: A loss function: $l(w,(x,y)) = \ln(1+\exp(-y\langle w,\tilde{x}\rangle))$

Parameter set $C = \{w \in \mathbb{R}^d : \|w\| \le 1\}$

Training set $S = \{s \in \mathbb{R}^{d-1} : \|x\| \le 1\}$

Number of iterations T = the size of training set.

Learning rate $\{\alpha_t : t = 1, \dots, T-1\}$. We set $\alpha_1 = \dots = \alpha_{T-1} = 0.1$.

1. Initialization: $W_1 = 0$
2. FOR $t = 1, \dots, T-1$
   a. Randomly pick a sample s_t from training set S
   b. Compute $G_t = \nabla l(W_t, s_t)$
   c. update: $W_{t+1} = \Pi_C(W_t - \alpha_t G_t)$
3. Return $\hat{w}_s = \frac{1}{T}\sum_{t=1}^{T} W_t$

2 Experiments:

The combinations of parameters for SGD including: learning rate = 0.1, training_size = 50; learning rate = 0.1, training_size = 100; learning rate = 0.1, training_size = 500; learning rate = 0.1, training_size = 1000.

During the experiment, we generized data with different σ values and training sizes, then computer the results respectively. Also, we used two arrays to store excess risks and mean errors for each scenario, so that we could plot the data as shown in Conclusion section below.

To generate the training and test datasets, we implemented two functions: generate_data and euclidean_projection.

The generate_data function first initializes the data as a 4*30 numpy array filled with initial values, and initialized the label as a 1*30 numpy array filled with initial values. Then the function iterates through the initialized dataset by assigning a random value between 0 and 1 to each data. If the random value is greater than ½, set the label to be -1, and use μ0 = (−1/4, −1/4, −1/4, −1/4) to generate the Gaussian vector u, then project u to X to get the desired data. If the random value is less than ½, set the label to be 1, and use μ1 = (1/4, 1/4, 1/4, 1/4) to generate the Gaussian vector u, then project u to X to get the desired data.
The euclidean_projection function compute the euclidean projection of a point to the parameter set C. We noticed that the feature space X and the parameter space C are both unit balls centered around the origin. Therefore after projection, the vector is actually a unit vector. Therefore, the projection process is actually the process of computing the unit vector. We use the numpy.linalg.norm function to computer the distance of the point to the origin, and then divide the point vector by the distance to get the projected vector.

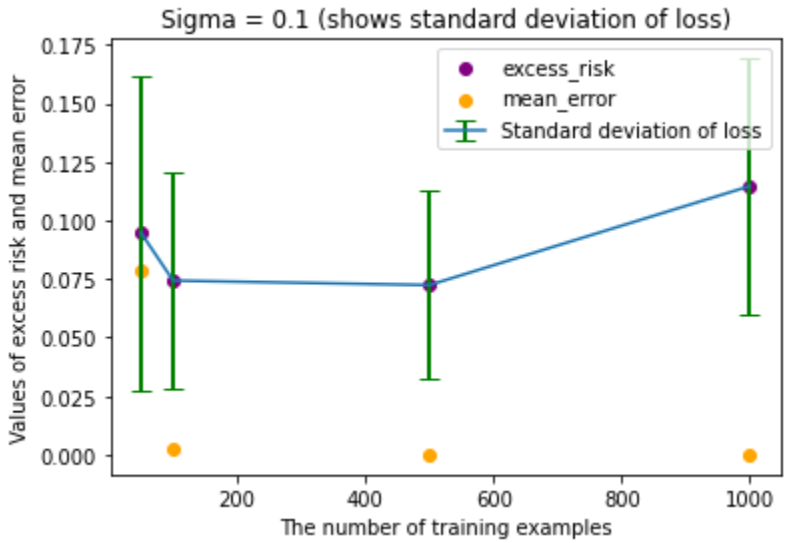
3 Analysis of Lipschitzness and Boundedness properties:

For logistic loss, the loss function l(w,(x,y)) is convex and ||x||-Lipschitz, therefore ρ = 5. The fact that the parameter set C is a unit ball deduces that the diameter of C is 2. Therefore M = 2.

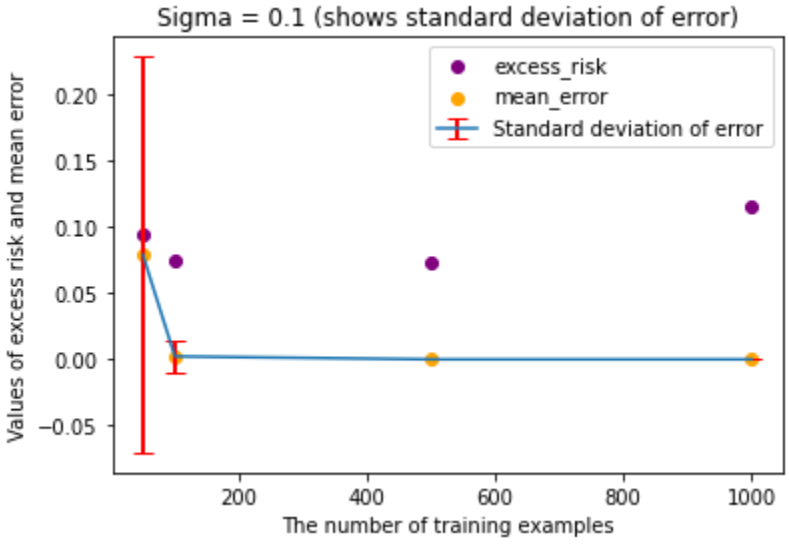
4 Results:

				Logistic loss				Classification error	
σ	n	N	# trials	Mean	Std Dev	Min	Excess Risk	Mean	Std Dev
0.1	50	400	30	0.56758	0.07372	0.47645	0.09112	0.10333	0.14866
0.1	100	400	30	0.51115	0.0584	0.4269	0.08426	0.00222	0.01197
0.1	500	400	30	0.46663	0.03763	0.38671	0.07993	0	0
0.1	1000	400	30	0.48644	0.0381	0.39842	0.08802	0	0
0.35	50	400	30	0.60012	0.06991	0.4639	0.13622	0.10222	0.11545
0.35	100	400	30	0.56681	0.14268	0.37882	0.18799	0.15667	0.02457
0.35	500	400	30	0.51126	0.14534	0.32805	0.18321	0.10556	0.01511
0.35	1000	400	30	0.49785	0.09768	0.33665	0.1612	0.36667	0.00999

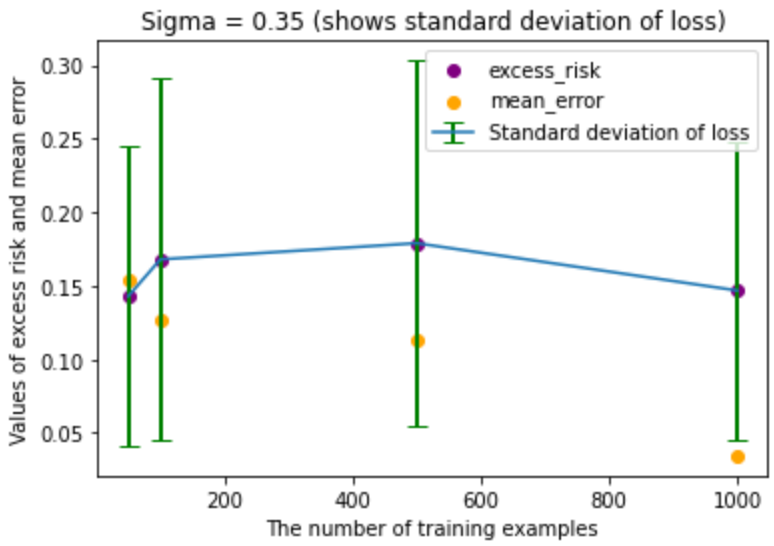
5 Conclusion:



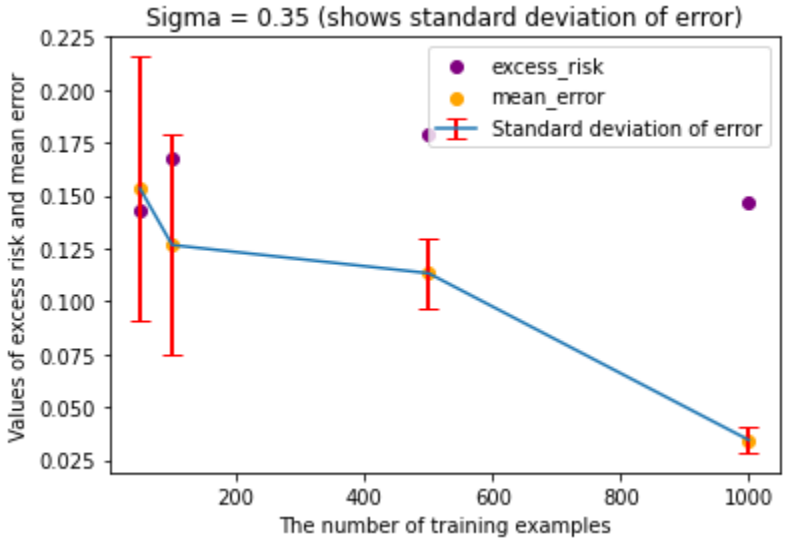
Plot 1: Sigma = 0.1 (shows standard deviation of loss)



Plot 2: Sigma = 0.1 (shows standard deviation of error)



Plot 3: Sigma = 0.35 (shows standard deviation of loss)



Plot 4: Sigma = 0.35 (shows standard deviation of loss)

The test results basically agree with the theoretical results. That is, after upating a set of parameters in an iterative way, the algorithm eventually reduces the value of the error function. When σ is 0.1, the value of the error function decreases faster than when σ is 0.35. It indicates that a dataset with smaller standard deviation is be easier to be learned by the SGD algorithm.This is because after each SGD algorithm run, the updated weight value will determine in what direction and with what magnitude the algorithm will move next time to be able to safely and quickly reduce the loss value towards the minimum loss value. Since the dataset features will be more scattered if the dataset is too scattered, the algorithm will need more iterations and self-correction time in the process of finding the optimal solution.

A Appendix: Symbol Listing

Symbol/Actual Name	Variable Name	Scope
numpy	np	Global
matplotlib.pylot	plt	Global
N	TESTING_SIZE	Global
x	data	generate_data function
y	labels	generate_data function
learning rate	eta	SGD function
		run function
n	training_size	run function

B Appendix: Library Routines

1. Numpy: A library that offers a wide range of computational functions. We imported this library as np.
2. Matplotlib: A library for making different types of plots. We imported this as plt.

C Appendix: Code

```
# Import libraries.
import numpy as np
import matplotlib.pyplot as plt

# Fix the size of testing data set as required.
TESTING_SIZE=30

# This function generates the data using the distribution introduced in
the project instruction.
def generate_data(size, sigma):
 data = np.zeros((4, size))
 labels = np.zeros((1, size))
 for x in range(0,size):
   if np.random.rand() > 0.5:
     data.T[x] = euclidean_projection(np.array([np.random.normal(-1/4,
sigma), np.random.normal(-1/4, sigma), np.random.normal(-1/4, sigma),
np.random.normal(-1/4, sigma)]))
```

```
      labels.T[x] = -1
    else:
      data.T[x] = euclidean_projection(np.array([np.random.normal(1/4,
sigma), np.random.normal(1/4, sigma), np.random.normal(1/4, sigma),
np.random.normal(1/4, sigma)]))
      labels.T[x] = 1
 data = np.vstack((np.ones((1, size)), data))
 return data, labels


# This function compute the euclidean projection of a point to the
parameter set C (which is specified in the project introduction).
def euclidean_projection(u):
 size = u.size
 origin = np.zeros(size)
 distance = np.linalg.norm(u - origin)
 x = u
 if distance > 1:
   x = u / np.linalg.norm(u)
 return x

# This function outputs the logistic loss given a dataset, the correct
label set of that dataset, and a weight vector.
def logistic_loss(w, x, y):
 return np.log(1 + np.exp(-y * np.dot(w, x)))

# This function outputs the gradient of logistic loss given a dataset, the
correct label set of that dataset, and a weight vector.
def logistic_loss_gradient(w, x, y):
 return (-y * x * np.exp(-y * np.dot(w, x))) / (1 + np.exp(-y * np.dot(w,
x)))

# This function is the implementation of SGD.
def SGD(data, labels, T, eta):
 w = np.zeros((T, 5)) # Create the array to store weight vertor for all
the iterations.
 for t in range(0,T-1):
   i = np.random.randint(data.shape[1]) # Randomly draw a data from the
dataset.
```

```
    G = logistic_loss_gradient(w[t], data.T[i], labels.T[i]) # Compute the
gradient.
    w[t + 1] = euclidean_projection((w[t].reshape(1, 5) - eta*G)) # Update
the weight.
  return np.mean(w, axis=0)

# This function runs the SGD and computes different metrics.
def run(eta,training_size, sigma, testing_data, testing_labels):
  trials = 30
  T = training_size
  training_data, training_labels = generate_data(training_size, sigma)
  w = np.zeros((trials, 5))
  mean_loss = np.zeros((trials, 1)) # Array to store the average of
logistic loss for each trial.
  std_loss = np.zeros((trials, 1)) # Array to store the standard deviation
of logistic loss for each trial.
  min_loss = np.zeros((trials, 1)) # Array to store the minimum value of
logistic loss for each trial.
  classification_error = np.zeros((trials, 1)) # Array to store the
classification error for each trial.

  for i in range(0,trials):
    w[i] = SGD(training_data, training_labels, T, eta)
    traing_loss = logistic_loss(w[i], training_data, training_labels)
    testing_loss = logistic_loss(w[i], testing_data, testing_labels)
    mean_loss[i] = np.average(testing_loss)
    std_loss[i] = np.std(testing_loss)
    min_loss[i] = np.min(testing_loss)

    # Compute the classification error.
    temp = np.sign(np.dot(w[i], testing_data)) - testing_labels
    errors = np.where(temp == 0, temp, 1) # Set the values that not equals
to 0 (2 and -2) to 1.
    classification_error[i] = np.average(errors)

  return np.mean(mean_loss), np.mean(std_loss), np.mean(min_loss),
np.mean(mean_loss-min_loss), np.mean(classification_error),
np.std(classification_error)
```