Project: Stochastic Gradient Descent

Yifei Zhang (.5387), Yue Lin (.3326)

November 2020

1 Introduction

The goal of this project is to implement stochastic gradient descent (SGD) algorithm and conduct a set of experiments considering two scenarios to evaluate the performance of SGD. The logistic loss and binary classification error are used in evaluation.

In this project, we implement an SGD using the following steps.

- Initialize the network weights $\mathbf{w}_1$ to 0 (ensure $\mathbf{w}_1 \in C$ for both scenarios).
- For each input feature vector $\tilde{\mathbf{x}}_t$ and its label y_t, calculate the gradient

$$\nabla f\left(\mathbf{w}_t\right) = \frac{-y_t \tilde{\mathbf{x}}_t \exp\left(-y_t \left\langle \mathbf{w}_t, \tilde{\mathbf{x}}_t \right\rangle\right)}{1 + \exp\left(-y_t \left\langle \mathbf{w}_t, \tilde{\mathbf{x}}_t \right\rangle\right)}$$

- Take a projected GD step

$$\mathbf{w}_{t+1} = \Pi_C\left(\mathbf{w}_t - \alpha \nabla f\left(\mathbf{w}_t\right)\right)$$

- Repeat steps 2 and 3 until all n samples are trained, and then output

$$\hat{\mathbf{w}} = \frac{1}{n} \sum_{t=1}^{n} \mathbf{w}_t$$

2 Experiments

This project is developed in Python. For each of the two scenarios, we test the experimental results using two $\sigma \in \{0.1, 0.35\}$ to generate the Gaussian distribution for both domain and parameter sets. For each σ value, we vary the size of the training set n as $50, 100, 500, 1000$. For each set of the parameters (σ, n), the following experiment is conducted.

- Calculate step size α based on the ρ-Lipschitz properties for each scenario described in the upcoming section.
- Generate data for model training and testing. For each example $(\mathbf{x}, y)$, y is randomly sampled from {-1,1} with equal probability. When $y = -1$, a 4-dimensional vector $\mathbf{u} = (u_1, u_2, u_3, u_4)$ is generated. Each element of $\mathbf{u}$ is randomly picked from Gaussian distribution with mean $-1/4$ and variance σ^2. When $y = 1$, each element of $\mathbf{u}$ is randomly picked from Gaussian distribution with mean 1/4 and variance σ^2(σ is specified above, $\sigma \in \{0.1, 0.35\}$).

- Perform Euclidean projection onto domain set χ,i.e $\mathbf{x} = \Pi_\chi(\mathbf{u})$.
  - Scenario 1: For $\mathbf{u}$ located outside of χ, each element u_i in $\mathbf{u}$ $\big(\forall j \in [4]\big)$ will be scaled into [-1,1] by choosing the closest value to u_i on χ, which is calculated as $sign\,(u_i) * min\,\big(|u_i|, 1\big)$. For $\mathbf{u}$ within χ, no projection is needed.
  - Scenario 2: For $\mathbf{u}$ located outside of χ, each element u_i in $\mathbf{u}$ will be shrunk so that the norm is 1 ,which is calculated as $\frac{u_i}{\|\mathbf{u}\|}$. For $\mathbf{u}$ within χ, no projection is needed.
- Run the SGD algorithm and evaluate the output $\hat{\mathbf{w}}$ on the test set using two types of metrics: logistic loss and classification error. The logistic loss is computed as

  $$l_{logist}\left(\mathbf{w}, (\mathbf{x}, y)\right) = \ln\Big(1 + \exp\left(-y\left\langle \mathbf{w}, \tilde{\mathbf{x}}\right\rangle\right)\Big)$$

  The classification error is computed as

  $$\mathbf{1}\Big(\text{sign}\left(\langle \mathbf{w}, \tilde{\mathbf{x}}\rangle\right) \neq y\Big)$$
- Repeat steps 2 and 3 for 30 trials (as determined in the project description). Calculate the below statistics for the specific (σ, n) setting.
  - Calculate the mean, minimum, and standard deviation of the 30 loss estimates. Calculate the difference between the mean and the minimum of the loss estimates as the expected excess risk.
  - Calculate the mean and standard deviation of the 30 binary classification error estimates.

3 Analysis of ρ-Lipschitz properties

For logistic loss function

$$l_{logist}\left(\mathbf{w}, (\mathbf{x}, y)\right) = \ln\Big(1 + \exp\left(-y\left\langle \mathbf{w}, \tilde{\mathbf{x}}\right\rangle\right)\Big)$$

where $\mathbf{x} \in \chi \subset \mathbb{R}^{d-1}$, $\tilde{\mathbf{x}} \triangleq (\mathbf{x}, 1)$, $y \in \{-1, +1\}$, $\mathbf{w} \in \mathbb{R}^d$, and $z = (\mathbf{x}, y)$.

Let's define $g_1\left(\mathbf{w}\right) \triangleq y\langle \mathbf{w}, \tilde{\mathbf{x}}\rangle$, $g_2\left(a\right) = \log\big(1 + \exp(-a)\big)$.

Note $l_{logist}\big(\cdot, (\mathbf{x}, y)\big) = g_2 \circ g_1$, g_1 is linear and $\|\tilde{\mathbf{x}}\|$-Lipschitz, g_2 is convex and 1-Lipschitz. Hence, $l_{logist}\big(\cdot, (\mathbf{x}, y)\big)$ is convex and $\|\tilde{\mathbf{x}}\|$-Lipschitz.

- For Scenario 1, $\|\tilde{\mathbf{x}}\|$ is bounded by $\rho = \sqrt{5}$.
  To prove C is a convex set, C is 5-dimensional hypercube in $\mathbb{R}^5$ i.e.

  $$C = \Big\{\mathbf{w} = (w_1, w_2, w_3, w_4, w_5) \in \mathbb{R}^5 : |w_j| \leq 1, \forall j \in [5]\Big\}$$

  We can re-write the above as $C = \big\{\mathbf{w} = (w_1, w_2, w_3, w_4, w_5) \in \mathbb{R}^5 : \|\mathbf{w}\|_\infty \leq 1\big\}$, where $\|\mathbf{w}\|_\infty = \underset{j \in [d}{max} |x_j|$.

  Let $\mathbf{a}, \mathbf{b} \in C$. For any $\lambda \in [0, 1]$, consider a vector $\mathbf{e}$ where

  $$\mathbf{e} \triangleq \lambda\mathbf{a} + (1 - \lambda)\,\mathbf{b}$$

We have

$$\begin{aligned}
\|\mathbf{e}\|_\infty &= \left\|\lambda\mathbf{a} + (1-\lambda)\,\mathbf{b}\right\|_\infty \\
&= \underset{j\in[5]}{max}|\lambda a_j + (1-\lambda)\,b_j| \\
&\le \lambda\underset{j\in[5]}{max}|a_j| + (1-\lambda)\underset{j\in[5]}{max}|b_j| \\
&\le \lambda\,\|\mathbf{a}\|_\infty + (1-\lambda)\,\|\mathbf{b}\|_\infty \\
&\le \lambda + (1-\lambda)\,(\mathbf{a},\mathbf{b}\in C) \\
&= 1
\end{aligned}$$

Hence, $\mathbf{e}\in C$. By the definition of convex set, C is convex set and bounded by $M = \sqrt{5*2^2} = 2\sqrt{5}$.

- For Scenario 2, $\|\tilde{\mathbf{x}}\|$ is bounded by $\rho = \sqrt{2}$.
  To prove C is a convex set, C is 5-dimensional unit ball in $\mathbb{R}^5$ i.e.

$$C = \left\{\mathbf{w}\in\mathbb{R}^5 : \|\mathbf{w}\|\le 1\right\}$$

Let $\mathbf{a},\mathbf{b}\in C$. For any $\lambda\in[0,1]$, consider a vector $\mathbf{e}$ where

$$\mathbf{e} \triangleq \lambda\mathbf{a} + (1-\lambda)\,\mathbf{b}$$

We have

$$\begin{aligned}
\|\mathbf{e}\| &= \left\|\lambda\mathbf{a} + (1-\lambda)\,\mathbf{b}\right\| \\
&\le \|\lambda\mathbf{a}\| + \left\|(1-\lambda)\,\mathbf{b}\right\| \\
&\le \lambda\,\|\mathbf{a}\| + (1-\lambda)\,\|\mathbf{b}\| \\
&\le \lambda + (1-\lambda)\,\big(\mathbf{a},\mathbf{b}\in C, \|\mathbf{a}\|\,, \|\mathbf{b}\|\le 1\big) \\
&= 1
\end{aligned}$$

Hence, $\mathbf{e}\in C$. By the definition of convex set, C is convex set and bounded by $M = 2$.

The learning rate α is constant for each scenario, which is calculated as

$$\alpha = \frac{M}{\rho\sqrt{T}}$$

Hence for scenario 1, $\alpha = \frac{2}{\sqrt{n}}$. For scenario 2, $\alpha = \sqrt{\frac{2}{n}}$.

4 Results

The experimental results are shown in Table 1. The expected excess risk and the standard deviation of risks can be found in Figure 1. The classification error and its standard deviation can be found in Figure 2.

In general, for two scenarios in different parameter settings, both excess risk and average classification error drop when n increases. When n takes larger values, such decrease seems less significant. The variances of risk and classification error follow the similar trend. The overall excess risk in scenario 1 is higher than that in scenario 2, but the classification error in scenario 1 is smaller than that in scenario 2.

					Logistic loss				Classification error	
Scenario	σ	n	N	#trials	Mean	Std Dev	Min	Excess Risk	Mean	Std Dev
1	0.10	50	30	0.418562	0.008753	0.403995	0.014566	0.001083	0.003962	0.234416
1	0.10	100	30	0.387852	0.003601	0.381059	0.006793	0.000083	0.000449	0.148509
1	0.10	500	30	0.347696	0.000734	0.346357	0.001339	0.000000	0.000000	0.004687
1	0.10	1000	30	0.339259	0.000708	0.338122	0.001137	0.000000	0.000000	0.000000
1	0.35	50	30	0.464567	0.017592	0.438874	0.025693	0.103500	0.033570	0.102299
1	0.35	100	30	0.427002	0.007682	0.413489	0.013512	0.096250	0.012890	0.131237
1	0.35	500	30	0.380281	0.002199	0.375819	0.004462	0.086250	0.007381	0.019663
1	0.35	1000	30	0.372902	0.001015	0.371356	0.001546	0.070083	0.002540	0.012599
2	0.10	50	30	0.517761	0.006741	0.508693	0.009068	0.015917	0.063392	0.253846
2	0.10	100	30	0.506215	0.002754	0.502531	0.003684	0.000083	0.000449	0.234262
2	0.10	500	30	0.488787	0.001267	0.487484	0.001302	0.000000	0.000000	0.022411
2	0.10	1000	30	0.488206	0.000882	0.487036	0.001169	0.000000	0.000000	0.000000
2	0.35	50	30	0.546855	0.010036	0.532957	0.013898	0.128500	0.065576	0.198564
2	0.35	100	30	0.534114	0.005336	0.527285	0.006829	0.094667	0.022265	0.172125
2	0.35	500	30	0.506864	0.001228	0.504717	0.002148	0.071583	0.005102	0.085103
2	0.35	1000	30	0.505185	0.000997	0.502768	0.002417	0.070000	0.003227	0.044888

Table 1: Experimental results.

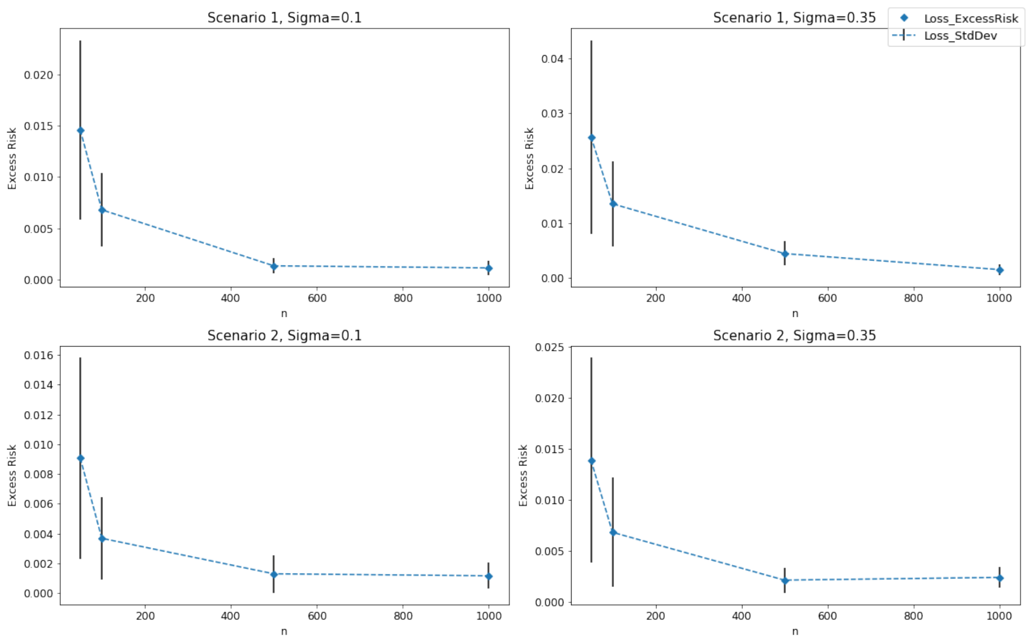
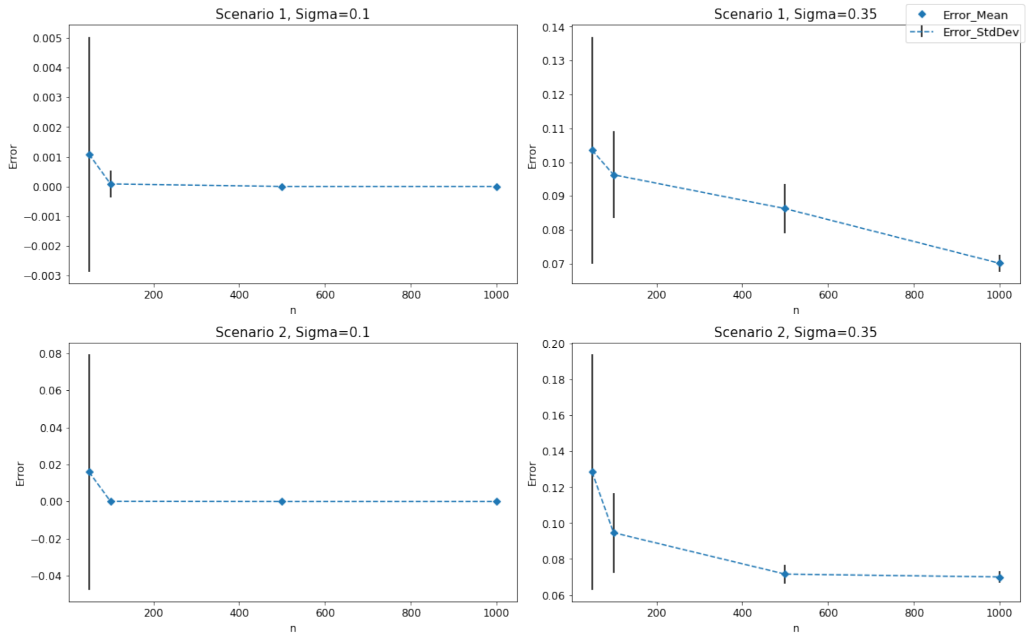


Figure 1: Expected excess risk and standard deviation of risks.

Figure 2: Classification error and its standard deviation.



5 Conclusion

- The experimental results agree with theoretical results.
  Based on the convergence properties of SGD, the excess risk is bound by $\frac{M\rho}{\sqrt{T}}$ (T= sample size n here). For $n = [50, 100, 500, 1000]$, the theoretical bound of scenario 1 is $[1.4142, 1.0, 0.4472, 0.3162]$, the theoretical bound of scenario 2 is $[0.4, 0.2828, 0.1265, 0.0894]$. For both scenarios, the theoretical bound of excess risk decreases as n increases. From the above table and figures, we can see that the experimental result exhibits the same trend as the theoretical result when n increases for both scenarios with $\sigma \in \{0.1, 0.35\}$. All the excess risk values fall into the theoretical bound.

- As n increases, the classification error generally decreases for both scenarios. This is because as we increase the size of the training set, the output $\hat{\mathbf{w}}$ is closer to $\mathbf{w}^* = \text{argmin} f(\mathbf{w})$, and therefore performs better on the test set.

- For each fixed n value, the excess risk in scenario 1 is larger than that in scenario 2. Based on the convergence properties of SGD, the bound is calculated by $\frac{M\rho}{\sqrt{T}}$. Given the equal $\sqrt{T}$ value, we know that $\frac{M\rho}{\sqrt{T}} = \frac{10}{\sqrt{T}}$ for scenario 1 is larger than $\frac{M\rho}{\sqrt{T}} = \frac{2\sqrt{2}}{\sqrt{T}}$ for scenario 2. Hence, theoretically the excess risk in scenario 2 should have tighter bound than that in scenario 1.

- When σ is increased from 0.1 to 0.35, both excess risk and binary classification error increase for both scenarios. This is because increasing the variance of the Gaussian distribution from which the training examples are drawn will add more randomness to the gradient derived in each step. This would affect the rate of convergence in SGD and therefore with the same number of training steps, the output $\hat{w}$ performs worse on the test set.

A Appendix: Symbol Listing

Symbol	Description	Variable name in code
$\mathbf{w}_t$	weight vector at t	w_t
$\tilde{\mathbf{x}}_t$	extended feature vector at t	X
y_t	label at t	y
$\nabla f(\mathbf{w}_t)$	gradient at t	g
χ	domain set	
C	parameter set	
Π_χ	euclidean projection onto χ	prj_data
Π_C	euclidean projection onto C	prj_grad
α	step size	l_rate
n	training set size	bs
N	test set size	test_n
$\mathbf{w}$	weight vector	w
$\mathbf{a}, \mathbf{b}, \mathbf{e}$	parameter vector in domain C	
$\mathbf{u}$	feature vector generated from Gaussian	
u_i	element in feature vector $\mathbf{u}$	
$\mathbf{x}$	projected feature vector	X
$\tilde{\mathbf{x}}$	extended feature vector of $\mathbf{x}$ (with '1' appended at the end of x)	X

B Appendix: Library Routines

Functions from `NumPy`:

- `numpy.linalg.norm`: calculate the vector norm; `ord` specifies the order of the vector
- `np.apply_along_axis`: apply a function to 1-D slices of an array along the given axis
- `np.random.normal`: draw random samples from a Gaussian distribution; `loc` specifies the mean of the distribution, and `scale` specifies the standard deviation of the distribution

C Appendix: Code

```
import numpy as np
import random
import matplotlib.pyplot as plt
import pandas as pd


def cube_prj(sample):
    '''
    This function projects both domain and parameter sets to a hypercube.

    sample: features or gradients, 1*d array (d: #dimension)

    return:
        the projected vector onto a hypercube with edge length 2 and centered around the origin
    '''
    return [np.sign(i) * min(np.abs(i), 1) for i in sample]


def ball_prj(sample):
    '''
    This function projects both domain and parameter sets to a unit ball.

    sample: features or gradients, 1*d array (d: #dimension)

    return:
        the projected vector onto a unit ball centered around the origin
    '''
    if np.linalg.norm(sample, ord=2) > 1:
        return sample / np.linalg.norm(sample, ord=2)
    else:
        return sample


def prj_data(X, y, prj_code):
    '''
    This function projects the domain set in terms for two scenarios.

    X: feature vectors, n*d array (n: #sample, d: #dimension)
    y: labels, 1*n array with values of -1 or +1
    prj_code: type of projection, 0 for cube, 1 for ball

    return:
        prj_x: projected feature vectors
        y: labels, same as the input
    '''
    if prj_code == 0:
        prj_x = np.apply_along_axis(cube_prj, 1, X)
    elif prj_code == 1:
        prj_x = np.apply_along_axis(ball_prj, 1, X)
    else:
        print("Please input correct code for projection type: 0 for cube, 1 for ball.")

    b = np.ones((prj_x.shape[0], 1))
    prj_x = np.append(prj_x, b, axis=1)
    return prj_x, y


def prj_grad(g, prj_code):
    '''
    This function projects the parameter set for two scenarios.

    g: gradients, 1*d array (d: #dimension)
    prj_code: type of projection, 0 for cube, 1 for ball

    return:
        prj_g: projected gradients
    '''
    if prj_code == 0:
        prj_g = cube_prj(g)
    elif prj_code == 1:
        prj_g = ball_prj(g)
    else:
        print("Please input correct code for projection type: 0 for cube, 1 for ball.")
    return prj_g


def gen_data(sig, n, d_dimension):
    '''
    This function generates the data for training and test.

    sig: standard deviation of the Gaussian function
    n: number of samples
    d_dimension: dimensionality of the feature vectors

    Return:
        X: feature vectors, n*d array (n: #sample, d: #dimension)
        y: labels, 1*n array with values of -1 and +1
    '''
    y = np.random.choice([-1, 1], p = [0.5, 0.5], size = n)
    X = np.array([])
    for i in range(n):
        if y[i] == -1:
            mu = -(1 / 4)
            negvec = np.random.normal(mu, sig, d_dimension)
            X = np.concatenate([X, negvec], axis=0)
        else:
            mu = (1 / 4)
            posvec = np.random.normal(mu, sig, d_dimension)
            X = np.concatenate([X, posvec], axis=0)
    X = np.reshape(X, (n, d_dimension))
    return X, y


def log_loss(X, y, w):
    '''
    This function outputs the logistic loss.

    X: feature vector, 1*d array (d: #dimension)
    y: label
    w: weight vector, 1*d array

    Return: logistic loss
    '''
    return np.log(1 + np.exp(-y * np.dot(w.T, X)))


def err(X, y, w):
    '''
    This function outputs the classification error.

    X: feature vector, 1*d array (d: #dimension)
    y: label
    w: weight vector, 1*d array

    Return: classification error
    '''
    return 0 if np.sign(np.dot(w.T, X)) == y else 1


def sdg(train_x, train_y):
    '''
    train_x: feature vectors one batch
    train_y: lables in one batch
    return:
            w: weight vector trained on one batch
    '''
    w_all = []
    w_t = np.zeros(train_x.shape[1])
    for idx in range(train_x.shape[0]):
        # Read data
        X = train_x[idx]
        y = train_y[idx]
        w_t = np.array(w_t)
        # Calculate gradient
        g = (-y * X * np.exp(-y * np.dot(w_t.T, X)) / (1 + np.exp(-y * np.dot(w_t.T, X))))
        # Project gradient
        w_t = prj_grad(np.add(w_t, np.multiply(-l_rate, g)), prj_code)
        # Backward propagation
        w_all.append(w_t)
    return np.average(np.array(w_all), axis=0)


def train(train_x, train_y, test_x, test_y, l_rate, n_epoch, bs, prj_code):
    '''
    This function implements and tests the SGD algorithm for logistic regression.

    train_x: feature vectors for training, n*d array (n: #sample, d: #dimension)
    train_y: labels for training, 1*n array
    test_x: feature vectors for test, n*d array (n: #sample, d: #dimension)
    test_y: labels for test, 1*n array
    l_rate: learning rate
    n_epoch: number of trials
    bs: training set size
    prj_code: type of projection, 0 for cube, 1 for ball

    Return:
        w: final weights
        risk_ave: average risk
        risk_min: minimum of all risks
        risk_var: standard deviation of all risks
        exp_excess_risk: expected excess risk
        cls_err_ave: average classification error
        cls_err_var: standard deviation of all classification errors
    '''
    risk_all = []
    cls_err_all = []

    for epoch in range(n_epoch):
        risk = cls_err = 0.
        train_x0 = train_x[epoch * bs: (epoch + 1) * bs] ## use current batch to for trainning
        train_y0 = train_y[epoch * bs: (epoch + 1) * bs] ## use current batch to for trainning
        w = sdg(train_x0, train_y0)

        # Evaluate
        for idx in range(test_x.shape[0]):
            # Read data
            X = test_x[idx]
            y = test_y[idx]
            # Evaluate
            risk += log_loss(X, y, w) / test_x.shape[0]
            cls_err += err(X, y, w) / test_x.shape[0]
        risk_all = np.append(risk_all, risk)
        cls_err_all = np.append(cls_err_all, cls_err)

    # Report risk
    risk_ave = np.average(risk_all)
    risk_min = np.amin(risk_all)
    risk_var = np.sqrt(np.var(risk_all))
    exp_excess_risk = risk_ave - risk_min
    # Report classification error
    cls_err_ave = np.average(cls_err_all)
    cls_err_var = np.sqrt(np.var(cls_err_all))
    return [w, risk_ave, risk_min, risk_var, exp_excess_risk, cls_err_ave, cls_err_var]


# Set up hyperparameters
n_epoch = 30    # training epochs
test_n = 400    # size of test set
d_dimension = 4
train_bs = np.array([50, 100, 500, 1000])  # batch size for each training epoch

np.random.seed(1)

result_list = []
for prj_code in [0, 1]:
    for sigma in [0.1, 0.35]:
        for bs in train_bs:

            if prj_code == 0:
                rho = np.sqrt(d_dimension + 1)
                m = 2 * np.sqrt(d_dimension + 1)
            else:
                rho = np.sqrt(2)
                m = 2

            l_rate = m / (rho * np.sqrt(bs))

            # Generate training data
            train_x, train_y = gen_data(sigma, bs * n_epoch, d_dimension)
            train_px, train_py = prj_data(train_x, train_y, prj_code)

            # Generate test data
            test_x, test_y = gen_data(sigma, test_n, d_dimension)
            test_px, test_py = prj_data(test_x, test_y, prj_code)

            # Train
            output = train(train_px, train_py, test_px, test_py, l_rate, n_epoch, bs, prj_code)
```

```
print('>scenario=%d, sigma=%.2f, n=%d, lr=%.2f, log_loss_mean=%.3f, \
    log_loss_std_dev=%.3f, log_loss_min=%.3f, \
    excess_risk=%.3f, cls_error_mean=%.3f, cls_error_std_dev=%.3f'
    % (prj_code + 1, sigma, bs, l_rate, output[1], output[3], \
        output[2], output[4], output[5], output[6]))
result = [prj_code + 1, sigma, bs, n_epoch,output[1], output[3], \
    output[2], output[4], output[5], output[6]]
result_list.append(result)
```

D Appendix: Acknowledgement

Yifei Zhang writes the code for data generation and checks the code of the entire project. Yue Lin writes the code for SGD. Yifei Zhang and Yue Lin draft the report together.