Project: Stochastic Gradient Descent

Yifei Zhang, Yue Lin

November 2020

1 Introduction

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

- Initialize the network weights $\mathbf{w}_1$ to random numbers between -1 and 1.
- For each training pattern $\tilde{\mathbf{x}}_t$ and its label y_t, calculate the gradient

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

- Project gradient $\nabla f(\mathbf{w}_t)$ to $\nabla \hat{f}(\mathbf{w}_t)$, then take a GD step

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

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

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

2 Experiments

3 Analysis of ρ-Lipschitz properties

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.

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

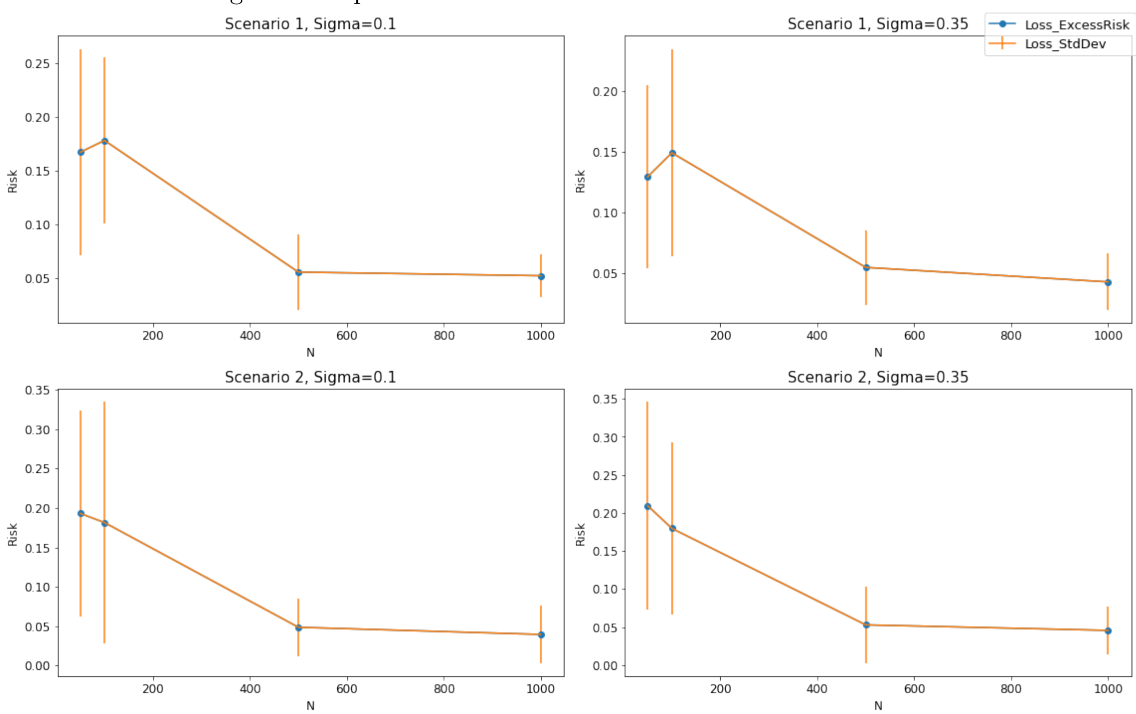
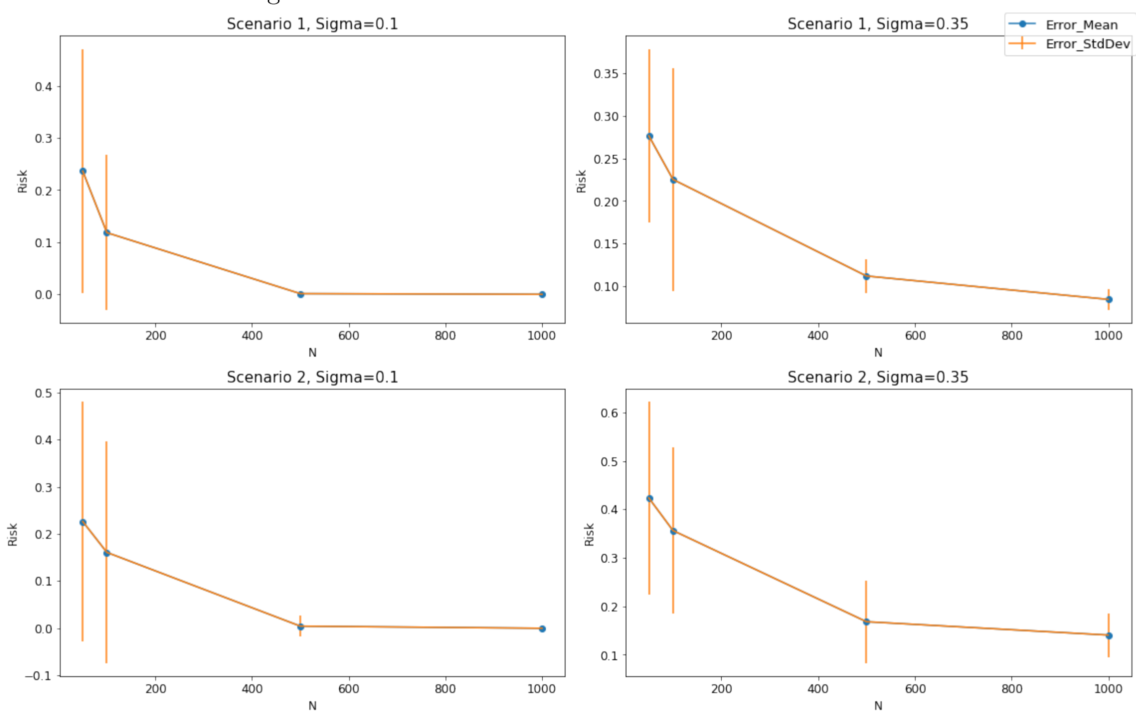


Figure 2: Classification error and its standard deviation.



					Logistic loss				Classification error	
Scenario	σ	n	N	#trials	Mean	Std Dev	Min	Excess Risk	Mean	Std Dev
1	0.1	50	400	30	0.549345	0.096150	0.382025	0.167320	0.236500	0.234416
1	0.1	100	400	30	0.536627	0.077158	0.358347	0.178280	0.118250	0.148509
1	0.1	500	400	30	0.421276	0.035334	0.365367	0.055909	0.001333	0.004687
1	0.1	1000	400	30	0.390512	0.020247	0.337979	0.052533	0.000000	0.000000
1	0.35	50	400	30	0.562977	0.075219	0.433738	0.129238	0.276667	0.102299
1	0.35	100	400	30	0.537027	0.085720	0.387975	0.149053	0.225333	0.131237
1	0.35	500	400	30	0.441465	0.030804	0.386903	0.054563	0.111833	0.019663
1	0.35	1000	400	30	0.418091	0.023252	0.375485	0.042606	0.084333	0.012599
2	0.1	50	400	30	0.534730	0.130990	0.341652	0.193078	0.226250	0.253846
2	0.1	100	400	30	0.518017	0.153447	0.336479	0.181538	0.160750	0.234262
2	0.1	500	400	30	0.388388	0.036358	0.339782	0.048606	0.004500	0.022411
2	0.1	1000	400	30	0.382979	0.036235	0.343476	0.039503	0.000000	0.000000
2	0.35	50	400	30	0.691756	0.136604	0.482097	0.209659	0.423500	0.198564
2	0.35	100	400	30	0.645894	0.113587	0.466219	0.179675	0.356250	0.172125
2	0.35	500	400	30	0.532771	0.050075	0.479885	0.052886	0.168250	0.085103
2	0.35	1000	400	30	0.522439	0.031785	0.476803	0.045636	0.140667	0.044888

Table 1: Experimental results.

5 Conclusion

A Appendix: Symbol Listing

$\mathbf{w}_t$	weights at t
$\tilde{\mathbf{x}}_t$	input pattern at t
y_t	label at t
$\nabla f(\mathbf{w}_t)$	gradient at t
$\nabla \hat{f}(\mathbf{w}_t)$	projected gradient at t
α	step size

B Appendix: Library Routines

C Appendix: Code

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

```python
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:
        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:
        a unit ball centered around the origin
    '''
    ratio = 1 / np.linalg.norm(sample)
    return [i * ratio for i in 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))
```

```python
    return X, y


def pred(X, w):
    '''
    This function makes binary classification using logistic regression.

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

    Return:
        yhat: predicted output
    '''
    yhat = 0.
    for i in range(X.shape[0]):
        yhat += w[i] * X[i]
    yhat = 1.0 / (1.0 + np.exp(-yhat))
    if yhat < 0.5:
        yhat = -1
    else:
        yhat = 1
    return yhat


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

    X: feature vectors, n*d array (n: #sample, d: #dimension)
    y: labels, 1*n array
    w: weight vectors, n*d array

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


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

    yhat: predicted label
    y: label

    Return: classification error
    '''
```

```
    if yhat == y:
        return 0
    else:
        return 1


def sgd(X, y, w_t, prj_code, l_rate):
    '''
    This function implements SGD.

    X: feature vectors, n*d array (n: #sample, d: #dimension)
    y: labels, 1*n array
    w_t: weights at t, n*d array
    prj_code: type of projection, 0 for cube, 1 for ball
    l_rate: learning rate

    Return:
        w_t: updated weight at t+1
    '''
    w_t = np.array(w_t)
    g = (-y * X * np.exp(-y * np.dot(w_t.T, X)) / (1 + np.exp(-y * np.dot(w_t.T, X))))
    w_t = prj_grad(np.add(w_t, np.multiply(-l_rate, g)), prj_code)
    return w_t


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
```

```python
    '''
    risk_all = []
    cls_err_all = []

    for epoch in range(n_epoch):
        w_t = np.random.uniform(-1, 1, (train_x.shape[1]))
        risk = cls_err = 0.
        w_all = []
        for idx in range(epoch * bs, (epoch + 1) * bs):
            # Read data
            X = train_x[idx]
            y = train_y[idx]
            # SGD
            w_t = sgd(X, y, w_t, prj_code, l_rate)
            # Backward propagation
            w_all.append(w_t)
        w = np.average(np.array(w_all), axis=0)

        # Evaluate
        for idx in range(test_x.shape[0]):
            # Read data
            X = test_x[idx]
            y = test_y[idx]
            # Predict
            yhat = pred(X, w)
            # Evaluate
            risk += log_loss(X, y, w) / test_x.shape[0]
            cls_err += err(yhat, y) / 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
```

```python
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:
                m = 2 * np.sqrt(d_dimension + 1)
            else:
                m = 2

            rho = d_dimension + 1
            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)
```