Feature Normalization

When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization (introduced in a later problem), features with larger values are treated as "more important", which is not usually what you want. One common approach to feature normalization is perform an affine transformation (i.e. shift and rescale) on each feature so that all feature values in the training set are in [0; 1]. Each feature gets its own transformation. We then apply the same transformations to each feature on the test1 set. It's important that the transformation is "learned" on the training set, and then applied to the test set. It is possible that some transformed test set values will lie outside the [0; 1] interval. Modify function $feature_normalization$ to normalize all the features to [0; 1]. (Can you use numpy's "broadcasting" here?) Note that a feature with constant value cannot be normalized in this way. Your function should discard features that are constant in the training set.

```
def feature_normalization(train, test):
    """Rescale the data so that each feature in the training set is in
    the interval [0,1], and apply the same transformations to the test
    set, using the statistics computed on the training set.

    Args:
        train − training set, a 2D numpy array of size (num_instances, num_features)
        test − test set, a 2D numpy array of size (num_instances, num_features)

    Returns:
        train_normalized − training set after normalization
        test_normalized − test set after normalization
    """
    # TODO
    train_min = np.min(train, axis = 0)
    train_max = np.max(train, axis = 0)
    train_range = train_max − train_min
    train_normalized = (train − train_min)/train_range
    test_normalized = (test − train_min)/train_range
    return train_normalized,test_normalized
```

Function 1: feature normalization

To do the feature normalization, we need to calculate the range of the numpy array, so compute the $train\_range$ and then

$$(train - train\_min)/train\_range$$

$$(test - train\_min)/train\_range$$

to finish the normalized.

Gradient Descent Setup

In linear regression, we consider the hypothesis space of linear functions $h_0 : \mathrm{R}^{\mathrm{d}} \to \mathrm{R}$, where

$$h_0(x) = \theta^T x$$

for $\theta, x \in \mathrm{R}^{\mathrm{d}}$, and we choose θ that minimizes the following "average square loss" objective function:

$$j(\theta) = 1/m * \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$$

where $(x_1, y_1), ..., (x_m, y_m) \in \mathrm{R}^{\mathrm{d}} \to \mathrm{R}$ is our training data. While this formulation of linear regression is very convenient, it's more standard to use a hypothesis space of "ane" functions:

$$h_{\theta,b}(x) = \theta^T x + b$$

which allows a "bias" or nonzero intercept term. The standard way to achieve this, while still maintaining the convenience of the rst representation, is to add an extra dimension to x that is always a xed value, such as 1. You should convince yourself that this is equivalent. We'll assume this representation, and thus we'll actually take $\theta, x \in \mathrm{R}^{\mathrm{d}}$.

1.Let $X \in \mathrm{R}^{\mathrm{m}*(\mathrm{d}+1)}$ be the **design matrix**, where the i th row of X is x_i, Let $y = (y_1, ..., y_m)^T \in \mathrm{R}^{\mathrm{m}} * 1$ be the "response". Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign.

$$\begin{aligned} j(\theta) &= 1/m * \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2 \\ &= \frac{2}{m}(X\theta - y)^T (X\theta - y) \end{aligned}$$

Or we can also write:

$$\begin{aligned} j(\theta) &= 1/2m * \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2 \\ &= \frac{1}{m}(X\theta - y)^T (X\theta - y) \end{aligned}$$

The 1m is to "average" the squared error over the number of components so that the number of components doesn't affect the function. The latter function $2J$ may seem more "natural," but the factor of 2 does not matter when optimizing. Can have better comparison for across model.

2.Write down an expression for the gradient of J (again, as a matrix/vector expression, without using an explicit summation sign).

$$j(\theta) = \frac{\partial(\frac{1}{m}(X\theta - y)^T(X\theta - y))}{\partial(X\theta - y)} \frac{\partial(X\theta - y)}{\partial\theta} \qquad = \frac{2}{m}(X\theta - y)^T X$$

Or we can also write:

$$j(\theta) = \frac{\partial(\frac{1}{2m}(X\theta - y)^T(X\theta - y))}{\partial(X\theta - y)} \frac{\partial(X\theta - y)}{\partial\theta} \qquad = \frac{1}{m}(X\theta - y)^T X$$

3.Use the gradient to write down an approximate expression for $J(\theta + \eta h) - J(\theta)$. The gradient at point θ is the best linear approximation of J at that point.

$$J(\theta + \eta h) - J(\theta) \approx \eta \nabla J(\theta) h$$

4.Write down the expression for updating in the gradient descent algorithm. Let be the step size.

$$\theta_{i+1} = \theta_i - \eta \nabla J(\theta), where \quad \eta > 0$$

5.Modify the function compute square loss

```
def compute_square_loss(X, y, theta):
    """
    Given a set of X, y, theta, compute the average square loss for predicting y with X*theta.
    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_features)
        y - the label vector, 1D numpy array of size (num_instances)
        theta - the parameter vector, 1D array of size (num_features)

    Returns:
        loss - the average square loss, scalar
    """
    loss = 0 #Initialize the average square loss
    #TODO
    y_predict = np.dot(X, theta)
    loss = np.dot(np.power(np.subtract(y_predict, y), 2), np.ones(X.shape[0])) / (2 * X.shape[0])
    return loss
# test with small dataset
X_small_data = np.random.rand(2,2)
y_small_data = np.random.rand(2,1)
theta_small_data = np.random.rand(2,1)
print(X_small_data)
print(y_small_data)
print(theta_small_data)
print(compute_square_loss(X_small_data, y_small_data, theta_small_data))
```

Function 2: compute square loss

6.Modify the function compute square loss gradient

```
def compute_square_loss_gradient(X, y, theta):
    """
    Compute the gradient of the average square loss (as defined in compute_square_loss), at the point
        ↪ theta.
    Args:
        X − the feature vector, 2D numpy array of size (num_instances, num_features)
        y − the label vector, 1D numpy array of size (num_instances)
            theta − the parameter vector, 1D numpy array of size (num_features)
    Returns:
        grad − gradient vector, 1D numpy array of size (num_features)
    """
    #TODO
    y_predict = np.dot(X, theta)
    loss_gradient = np.dot(np.subtract(y_predict, y), X) / X.shape[0]
    return loss_gradient

theta = np.ones(X_train.shape[1])
loss = compute_square_loss(X_train, y_train, theta)
loss_gradient = compute_square_loss_gradient(X_train, y_train, theta)
print(loss)
print(loss_gradient)
# test with small dataset
X_small_data = np.random.rand(2,2)
y_small_data = np.random.rand(2,1)
theta_small_data = np.random.rand(2,1)
print(X_small_data)
print(y_small_data)
print(theta_small_data)
print(compute_square_loss(X_small_data, y_small_data, theta_small_data))
print(compute_square_loss_gradient(X_small_data, y_small_data, theta_small_data))
```

Function 3: compute square loss gradient and run both functions

```
In [7]: X_small_data = np.random.rand(2,2)
   ...: y_small_data = np.random.rand(2,1)
   ...: theta_small_data = np.random.rand(2,1)
   ...: print(X_small_data)
   ...: print(y_small_data)
   ...: print(theta_small_data)
   ...: print(compute_square_loss(X_small_data, y_small_data, theta_small_data))
   ...: print(compute_square_loss_gradient(X_small_data, y_small_data, theta_small_data))
[[0.04481616 0.96389959]
 [0.34460268 0.7379267 ]]
[[0.66790317]
 [0.27568372]]
[[0.50625769]
 [0.59607357]]
0.029916209809554204
[[0.05676351]
 [0.09088866]]
```

Figure 1: small dataset to test square loss and loss gradient

Name	Type	Size	Value
X	float64	(200, 48)	[[0.00000000e+00 0.00000000e+00 0.00000000e+00 ... -7.81824199e-01 ...
X_test	float64	(100, 49)	[[1. 1. 1. ... 0.21229954 0.21229954 1. ...
X_train	float64	(100, 49)	[[1. 1. 1. ... 0.13241261 0.13241261 1. ...
df	DataFrame	(200, 49)	Column names: x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x ...
loss	float64	1	441.4342610243893
loss_gradient	float64	(49,)	[26.07013205 25.58326502 25.05054258 ... 16.85255853 1… 26.2 ...
theta	float64	(49,)	[1. 1. 1. ... 1. 1. 1.]
x_test	float64	(100, 48)	[[1. 1. 1. ... -0.61692371 … -3 ...
x_train	float64	(100, 48)	[[1. 1. 1. ... -0.67897469 … -3 ...
y	float64	(200,)	[-1.37657523 0.87878245 1.10870068 ... 3.37935567 … 3.9 ...
y_test	float64	(100,)	[-1.07263709 -1.91105204 -1.07036941 ... -2.95765686 -… 4.2 ...
[illegible]	[illegible]	[illegible]	[-2.79297326 4.73657935 1.43620799 ... -3.4037511 …

Figure 2: import data

```
loading the dataset
Split into Train and Test
Scaling all to [0, 1]
441.4342610243893
[26.07013205 25.58326502 25.05054258 24.74081471 24.24546864 23.79891226
 22.90863314 22.90863314 21.90026135 20.0569049  18.36416632 17.3017451
 14.5002761  12.45800123  8.21040281  7.34556074  5.24340575  0.90770525
 23.14742097 23.14742097 23.14742097 21.48832491 21.48832491 21.48832491
 19.8972414  19.8972414  19.8972414  19.15895467 19.15895467 19.15895467
 18.74898619 18.74898619 18.74898619 11.85229611 11.85229611 11.85229611
 14.39885467 14.39885467 14.39885467 15.83930209 15.83930209 15.83930209
 16.49353617 16.49353617 16.49353617 16.85255853 16.85255853 16.85255853
 26.23768143]
```

Figure 3: loss and gradient loss result

Batch Gradient Descent

At the end of the skeleton code, the data is loaded, split into a training and test set, and normalized. We'll now finish the job of running regression on the training set. Later on we'll plot the results together with SGD results.

1.Complete batch gradient descent.

```
def batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=False):
    """
    In this question you will implement batch gradient descent to
    minimize the average square loss objective.

    Args:
        X - the feature vector, 2D numpy array of size (num_instances, num_features)
        y - the label vector, 1D numpy array of size (num_instances)
        alpha - step size in gradient descent
        num_step - number of steps to run
        grad_check - a boolean value indicating whether checking the gradient when updating

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array of size (num_step+1,
            ↪ num_features)
                for instance, theta in step 0 should be theta_hist[0], theta in step (num_step) is
                    ↪ theta_hist[-1]
        loss_hist - the history of average square loss on the data, 1D numpy array, (num_step+1)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    theta = np.zeros(num_features) #Initialize theta
    #TODO
    loss_hist[0] = compute_square_loss(X, y, theta) # initial loss
    theta_hist[0] = theta # initial theta
    for i in range(num_step):
        if grad_check:
            grad = grad_checker(X, y, theta.T)
            print('Grade check:{0}'.format(grad))

        loss_gradient = compute_square_loss_gradient(X, y, theta.T)
        theta = theta - alpha*loss_gradient.T
        theta_hist[i+1] = theta
        loss = compute_square_loss(X, y, theta.T)
        loss_hist[i+1] = loss

    return theta_hist, loss_hist
```

Function 4: batch gradient descent function

loss_return	float64	(1001, 49)	[[0. 0. 0. ... 0. … 0. ...
theta	float64	(49,)	[1. 1. 1. ... 1. 1. 1.]
theta_return	float64	(1001,)	[3.98075917 3.91325976 3.85188424 ... 1.03900245 1.03892598 1.03884969 ...

Figure 4: data result for batch gradient algo

```
loss return: [[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [-0.01259103  0.00750755  0.02025476 ...  0.01444064  0.01444064
  -0.01848617]
 [-0.05580238 -0.01564154  0.00998855 ...  0.00770169  0.00770169
  -0.06757517]
 ...
 [-1.5671494   0.64936919  1.43509585 ... -0.00771457 -0.00771457
  -1.16289895]
 [-1.56727897  0.649472    1.43460544 ... -0.00782174 -0.00782174
  -1.16264481]
 [-1.56740818  0.64957502  1.43411528 ... -0.00792888 -0.00792888
  -1.16239102]]
theta return: [3.98075917 3.91325976 3.85188424 ... 1.03900245 1.03892598
1.03884969]
```

Figure 5: result for batch gradient

2.Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge. Starting with a step-size of 0:1, try various different fixed step sizes to see which converges most quickly and/or which diverge. As a minimum, try step sizes 0.5, 0.1, .05, and .01. Plot the average square loss as a function of the number of steps for each step size. Briefly summarize your findings.

```
def step_size_plot(X, y):
    alphas = [0.01, 0.05, 0.1, 0.5]
    color = ["red", "green","blue", "yellow"]
    for i,alpha in enumerate(alphas):
        theta_hist, loss_hist = batch_grad_descent(X, y, alpha, num_step=1000)
        plt.plot(loss_hist, label=f"Alpha={alpha}", color = color[i])
    plt.yscale('log')
    plt.xlabel('Steps')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

Function 5: plot for log loss with different alpha

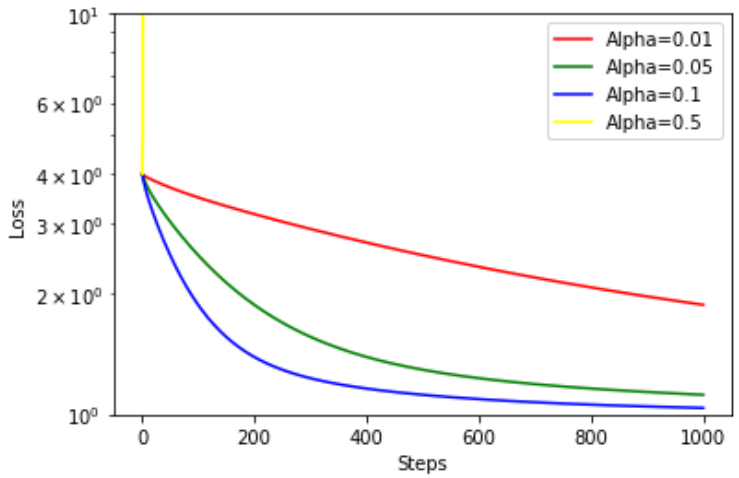

Figure 6: plot for log loss with different alpha

The solution we can conclude from the plot is the larger the alpha, the converge will faster.

3.Implement backtracking line search. How does it compare to the best fixed step-size you found in terms of number of steps? In terms of time? How does the extra time to run backtracking line search at each step compare to the time it takes to compute the gradient? (You can also compare the operation counts.)

```
def backtracking_line_search(X, y, theta):
    sigma = 0.01
    beta = 0.5
    k = 0
    alpha = 1 #step_size
    while(True):
        loss_gradient = compute_square_loss_gradient(X,y,theta)
        theta_next = theta - alpha * loss_gradient
        if (compute_square_loss(X,y,theta) - compute_square_loss(X,y,theta_next) >= sigma * alpha
            ↪ * np.dot(loss_gradient.T,loss_gradient)):
            break
        else:
            alpha = beta * alpha
            theta = theta_next
            k = k+1
    print(alpha)
def step_size_plot(X, y):
    alphas = [0.01, 0.05, 0.1, 0.5, 0.0625]
    color = ["red", "green","blue", "yellow", "pink"]
    for i,alpha in enumerate(alphas):
        start = time.time()
        theta_hist, loss_hist = batch_grad_descent(X, y, alpha, num_step=1000)
        end = time.time()
        print("the" + str(i) + "alpha test cost:" + str(end - start))
        plt.plot(loss_hist, label=f"Alpha={alpha}", color = color[i])
    plt.yscale('log')
    plt.xlabel('Steps')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

Function 6: backtracking line search

```
In [90]: backtracking_line_search(X_train,y_train,np.ones(X_train.shape[1]))
0.0625
```

Figure 7: alpha after backtracking line search

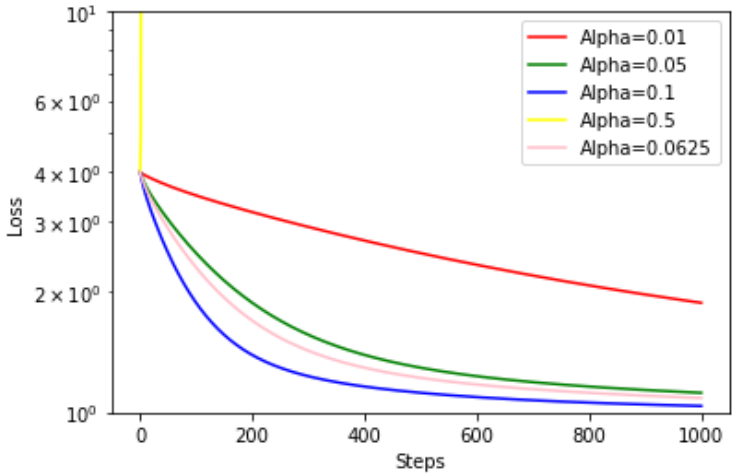

Figure 8: backtracking result is 0.0625 in pink line

```
the0alpha test cost:0.01694035530090332
the1alpha test cost:0.015965700149536133
the2alpha test cost:0.015947580337524414
the3alpha test cost:0.016998291015625
the4alpha test cost:0.016953468322753906
```

Figure 9: alpha after backtracking line search

The pink line is for backtracking result, that one converges in less steps, but only sightly different after 100 steps. And when I use time.time() to record the time for each different alphas, there are not too big difference between them.

Stochastic Gradient Descent

When the training data set is very large, evaluating the gradient of the objective function can take a long time, since it requires looking at each training example to take a single gradient step. When the objective function takes the form of an average of many values, such as

$$J(\theta) = 1/m \sum_{i=1}^{m} f_i(\theta)$$

1.Show that the objective function $\mathrm{j}(\theta) = 1/m * \sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2$can be writtern in the form $j(\theta) = 1/m * \sum_{i=1}^{m} f_i(\theta)$ by giving an expression for $f_i(\theta)$ that makes the two expressions equivalent.
I think the function should be $j(\theta) = 1/m * \sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta$.

$$\begin{aligned} j(\theta) &= 1/m * \sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta \\ &= 1/m * \sum_{i=1}^{m}[(h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta] \qquad = (h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta \end{aligned}$$

Or if we don't contain $\lambda\theta^T\theta$, then the answer is $(h_\theta(x_i) - y_i)^2$.

2.Show that the stochastic gradient

$$\mathbb{E}_i \nabla f_i(x) = 1/m \sum_{i=1}^{m} \nabla f_i(x) = \nabla f(x)$$

3.Write down the update rule for θ in SGD for the linear regression objective function.

$$\theta_{t+1} = \theta_t - \eta[2/m(\theta_i^T x_i - y_i)x_i + 2\lambda\theta_i]$$

Or we can write

$$\theta_{t+1} = \theta_t - \eta(2/m(\theta_i^T x_i - y_i)x_i)$$

4.Implement stochastic grad descent.

```
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg = 0.01, num_epoch=1000):
    """
    In this question you will implement stochastic gradient descent with regularization term

    Args:
        X − the feature vector, 2D numpy array of size (num_instances, num_features)
        y − the label vector, 1D numpy array of size (num_instances)
        alpha − string or float, step size in gradient descent
                NOTE: In SGD, it's not a good idea to use a fixed step size. Usually it's set to 1/sqrt(t) or
                    ↪ 1/t
                if alpha is a float, then the step size in every step is the float.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t).
                if alpha == "1/t", alpha = 1/t.
        num_epoch − number of epochs to go through the whole training set

    Returns:
        theta_hist − the history of parameter vector, 3D numpy array of size (num_epoch,
            ↪ num_instances, num_features)
                    for instance, theta in epoch 0 should be theta_hist[0], theta in epoch (num_epoch) is
                        ↪ theta_hist[−1]
        loss hist − the history of loss function vector, 2D numpy array of size (num_epoch,
            ↪ num_instances)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) #Initialize theta
    theta_hist = np.zeros((num_epoch, num_instances, num_features)) #Initialize theta_hist
    loss_hist = np.zeros((num_epoch, num_instances)) #Initialize loss_hist
    #TODO
    step_size = 0
    for i in range(num_epoch):

        for j in range(num_instances):
            theta_hist[i,j] = theta
            regulariztion_loss = lambda_reg * np.dot(theta.T,theta)
            loss_hist[i,j] = compute_square_loss(X, y, theta) + regulariztion_loss
            gradient_part = 2*(np.dot(theta.T, X[j])−y[j])*X[j] + 2*lambda_reg*theta

            if alpha == "1/sqrt(t)":
                theta = theta − 0.1/step_size *gradient_part
            elif alpha == "1/t":
                theta = theta − 0.1/np.sqrt(step_size)*gradient_part
            else:
                theta = theta − alpha*gradient_part
            step_size += 1
    return theta_hist, loss_hist
```

Function 7: stochastic gradient descent

loss_gradient	float64	(49,)	[26.07013205 25.58326502 25.05054258 ... 16.85255853 16.85255853 26.2 ...
loss_hist	float64	(1000, 100)	[[441.92426102 396.7787976 18.63538062 ... 3.30285244 3.30271564 ...
loss_return	float64	(1001, 49)	[[0. 0. 0. ... 0. 0. 0. ...
theta	float64	(49,)	[1. 1. 1. ... 1. 1. 1.]
theta_hist	float64	(1000, 100, 49)	[[[1. 1. 1. ... 1. 1. 1. ...
theta_return	float64	(1001,)	[3.98075917 3.91325976 3.85188424 ... 1.03900245 1.03892598 1.03884969 ...

Figure 10: SGD output

5. In this case we are investigating the convergence rate of the optimization algorithm with dierent step size schedules, thus we're interested in the value of the objective function, which includes the regularization term. Sometimes the initial step size $CforC/tandC/\sqrt{t}$ is too aggressive and will get you into a part of parameter space from which you can't recover. Try reducing C to counter this problem.

I plot the data for different alpha, ["$1/t$","$1/sqrt(t)$", $0.0005, 0.001, 0.01$], and when the step size equal to $0.1/sqrt(t)$ performs the best. but the step size equal to $0.1/t$ does not converge. (Because I define the step_size to $0.1/sqrt(t)$ if the enter alpha is $1/sqrt(t)$. Same with $1/t$)

```
def test_SGD(X, y):
    alphas = ["1/t","1/sqrt(t)",0.0005,0.001, 0.01]
    fig = plt.figure(figsize = (20,8))
    plt.subplot(222)
    for alpha in alphas:
        [theta_hist, loss_hist] = stochastic_grad_descent(X, y, alpha, num_epoch = 5)
        plt.plot(np.log(loss_hist.ravel()), label = 'alpha:'+str(alpha))
    plt.legend()
```

Function 8: plot the SGD with different alpha

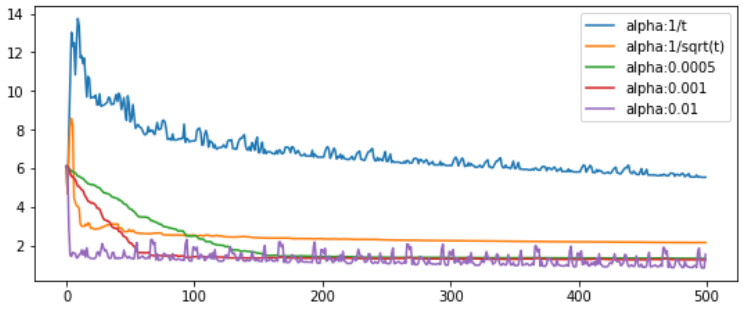

Figure 11: SGD plot