Machine Learning and Computational Statistics
Homework 1: Ridge Regression, Gradient Descent, and SGD

1 Introduction

2 Linear Regression

2.1 Feature Normalization

```
def feature_normalization(train, test):
  train_normalized = (train - np.amin(train,axis = 0)) /
   ↪  (np.amax(train,axis = 0) - np.amin(train,axis = 0))
  test_normalized = (test - np.amin(train,axis = 0)) /
   ↪  (np.amax(train,axis = 0) - np.amin(train,axis = 0))
  return (train_normalized, test_normalized)
```

2.2 Gradient Descent Setup

2.2.1

$$J(\theta) = \frac{1}{m}\|y - X\theta\|_2^{\,2} = \frac{1}{m}(y - X\theta)^T(y - X\theta)) = \frac{1}{m}(y^T y - 2y^T X\theta + \theta^T X^T X\theta)$$

2.2.2

$$\nabla_\theta J(x,\theta) = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \dots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix} = \frac{1}{m}(-2X^T y + 2X^T X\theta)$$

2.2.3

$$J(\theta + \eta h) - J(\theta) = \nabla_x J(x, \theta)^T \eta h = \frac{1}{m}(-2X^T y + 2X^T X\theta)^T \eta h = \frac{1}{m}(-2y^T X + 2\theta^T X^T X)\eta h$$

2.2.4

$$\theta_{k+1} = \theta_k - \eta \nabla_x J(x, \theta) = \theta_k - \frac{1}{m}(-2X^T y + 2X^T X\theta)\eta$$

2.2.5

```
def compute_square_loss(X, y, theta):
  num_instances, num_features = X.shape[0], X.shape[1]
  loss = 0
  l = y - X.dot(theta)
  loss = sum(i*i for i in l)/num_instances
  return loss
```

2.2.6

```
def compute_square_loss_gradient(X, y, theta):
  num_instances, num_features = X.shape[0], X.shape[1]
  grad = -2*(y-X.dot(theta)).dot(X)/num_instances
  return grad
```

2.3 (OPTIONAL) Gradient Checker

```
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):

  true_gradient = compute_square_loss_gradient(X, y, theta) #the true
   ↪ gradient
  num_features = theta.shape[0]
  approx_grad = np.zeros(num_features)

  for i in range(num_features):
      theta_plus = np.copy(theta)
      theta_minus = np.copy(theta)
      theta_plus[i] = theta_plus[i] + epsilon
      theta_minus[i] -= epsilon
      ag = (compute_square_loss(X, y, theta_plus) -
      compute_square_loss(X, y, theta_minus))/(2*epsilon)
      approx_grad[i] = ag
      if abs(approx_grad[i] - true_gradient[i]) > tolerance:
          return False

  return True


def generic_gradient_checker(X, y, theta, objective_func,
adient_func, epsilon=0.01, tolerance=1e-4):

  true_gradient = gradient_func(X, y, theta) #the true gradient
  num_features = theta.shape[0]
  approx_grad = np.zeros(num_features)

  for i in range(num_features):
      theta_plus = np.copy(theta)
      theta_minus = np.copy(theta)
      theta_plus[i] = theta_plus[i] + epsilon
      theta_minus[i] -= epsilon
      ag = (objective_func(X, y, theta_plus) - objective_func(X, y,
       ↪ theta_minus))/(2*epsilon)
      approx_grad[i] = ag
      if abs(approx_grad[i] - true_gradient[i]) > tolerance:
          return False

  return True
```

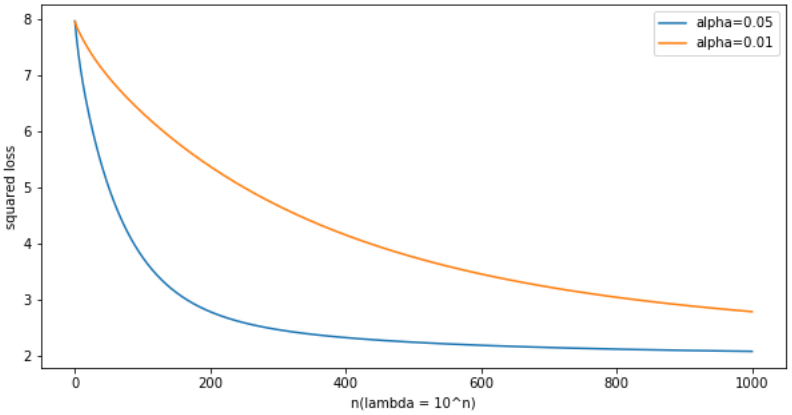
2.4 Batch Gradient Descent

2.4.1

```python
def batch_grad_descent(X, y, alpha=0.1, num_iter=1000, check_gradient=False, stop_if = 0, stop = 0.01):
    """
    In this question you will implement batch gradient descent to minimize the 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_iter - number of iterations to run
        check_gradient - a boolean value indicating whether checking the gradient when updating
        stop_if - whether to use stop criteria
        stop - stop criteria: if the difference between two iteration is smaller than stop, then break and return

    Returns:
        theta_hist - store the the history of parameter vector in iteration, 2D numpy array of size (num_iter+1, num_features)
                     for instance, theta in iteration 0 should be theta_hist[0], theta in ieration (num_iter) is theta_hist[-1]
        loss_hist - the history of objective function vector, 1D numpy array of size (num_iter+1)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_iter+1, num_features))  #Initialize theta_hist
    loss_hist = np.zeros(num_iter+1) #initialize loss_hist
    theta = np.zeros(num_features) #initialize theta
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta)
    if check_gradient == True & grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4) == False:
        return False
    for i in range(num_iter):
        if check_gradient == True and not grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
            return False
        gradient = compute_square_loss_gradient(X, y, theta)
        theta = theta - alpha * gradient
```

```
        theta_hist[i+1] = theta
        loss_hist[i+1] = compute_square_loss(X, y, theta)
        if stop_if == 1:
            if loss_hist[i+1]-loss_hist[i-1] < stop:
                break
    return theta_hist, loss_hist
```

2.4.2

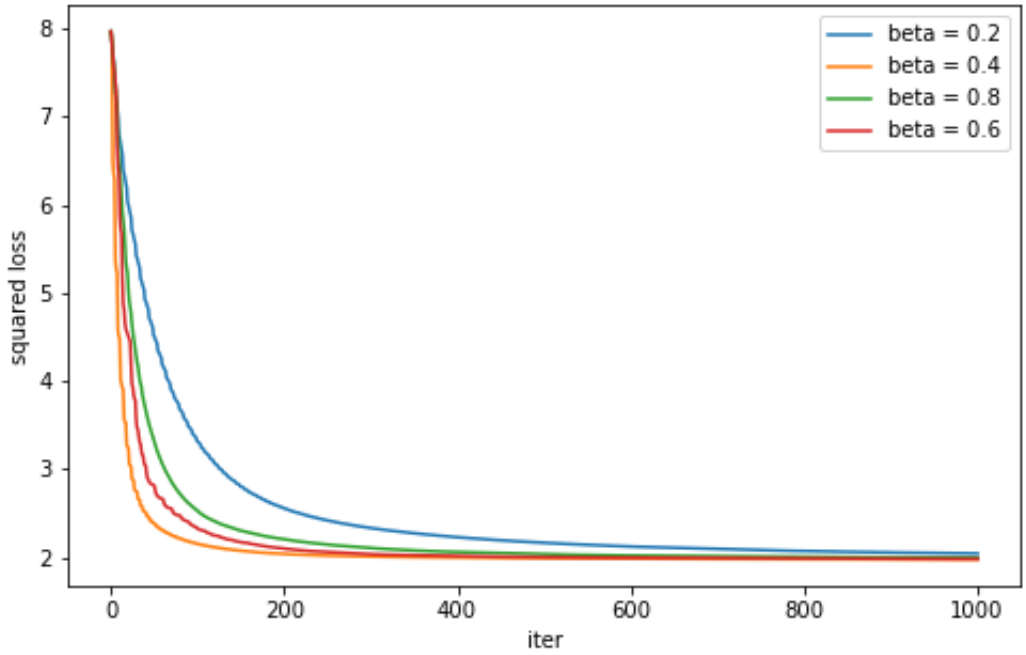


For four step sizes [0.01, 0.05, 0.1, 0.5], only 0.01 and 0.05 converge, 0.1, 0.5 both don't converge. For 0.5, 0.1, the loss function goes beyond limit in 200 steps. For 0.01 and 0.05, the loss goes down with the increase of the steps. The function converge much faster using step size 0.05 than that using step size 0.01.

2.4.3

```
def batch_grad_descent_back(X, y, beta = 0.4, alpha=0.1, num_iter=1000, check_gradient=False, stop_if = 0, stop = 0.01):
    '''
    Implement backtracking line search to do batch gradient descent to minimize the 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_iter - number of iterations to run
        check_gradient - a boolean value indicating whether checking the gradient when updating
        stop_if - whether to use stop criteria
        stop - stop criteria: if the difference between two iteration is smaller than stop, then break and return
    Returns:
        theta_hist - store the the history of parameter vector in iteration, 2D numpy array of size (num_iter+1, num_features)
                     for instance, theta in iteration 0 should be theta_hist[0], theta in ieration (num_iter) is theta_hist[-1]
        loss_hist - the history of objective function vector, 1D numpy array of size (num_iter+1)
    """
    '''
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_iter+1, num_features))  #Initialize theta_hist
    loss_hist = np.zeros(num_iter+1) #initialize loss_hist
    theta = np.zeros(num_features) #initialize theta
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta)
    temp = alpha
    for i in range(num_iter):
        alpha = temp
        gradient = compute_square_loss_gradient(X, y, theta)
        while (compute_square_loss(X, y, theta - alpha * gradient) > compute_square_loss(X, y, theta) - alpha / 2 * np.sum([gr**2 for gr in gradient])):
            alpha *= beta
        theta = theta - alpha * gradient
        theta_hist[i+1] = theta
        loss_hist[i+1] = compute_square_loss(X, y, theta)
        if stop_if == 1:
```

```
            if loss_hist[i+1]-loss_hist[i-1] < stop:
                break
    return theta_hist, loss_hist
```



The above graph shows the graph for backtracking line search using beta = 0.8, 0.6, 0.4, 0.2. The backtracking line search converge much faster than the batch gradient descent in terms of step size.

I test each method using the same stop criteria. The time for each beta is [0.0025839805603027344, 0.0013079643249511719, 0.0006451606750488281, 0.0005328655242919922], whereas the time for each alpha for Batch Gradient Descent is [0.051631927490234375, 0.03989100456237793]. The backtracking line search converge much faster than the batch gradient descent in terms of time.

We can see that the operation count to compute whether to change alpha is bigger than that to update alpha. Therefore the extra time to run backtracking line search at each step is longer than the time it takes to compute the gradient when comparing operation counts.

2.5 Ridge Regression (i.e. Linear Regression with ℓ_2 regularization)

2.5.1 1

$$\nabla_x J(\theta) = \frac{1}{m}(-2X^T y + 2X^T X\theta) + 2\lambda\theta$$

$$\theta = \theta - \eta\nabla_x J(\theta) = \theta - (\frac{1}{m}(-2X^T y + 2X^T X\theta) + 2\lambda\theta)\eta$$

2.5.2 2

```
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    """
    Compute the gradient of L2-regularized square loss function given X, y and 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)
        lambda_reg - the regularization coefficient

    Returns:
        grad - gradient vector, 1D numpy array of size (num_features)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    grad = -2*((y-X.dot(theta)).dot(X))/num_instances+2*lambda_reg * theta
    return grad
```

2.5.3 3

```
def regularized_grad_descent(X, y, alpha=0.1, lambda_reg=1, num_iter=1000):
    """
    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
        lambda_reg - the regularization coefficient
        numIter - number of iterations to run

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array of size (num_iter+1, num_features)
        loss_hist - the history of loss function without the regularization term, 1D numpy array.
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_iter+1, num_features))  #Initialize theta_hist
    loss_hist = np.zeros(num_iter+1) #initialize loss_hist
    theta = np.zeros(num_features) #initialize theta
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta)
    for i in range(num_iter):
        gradient = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)
        theta = theta - alpha * gradient
        theta_hist[i+1] = theta
        loss_hist[i+1] = compute_square_loss(X, y, theta)
    return theta_hist, loss_hist
```

2.5.4 4

If we use a very large number B for the extra bias dimension, the parameter for the bias term will be very small, therefore making the regularization as weak as we like.

2.5.5 5

Making B larger will decrease the effective regularization on the bias term λB^2, and the effective regularization on the bias term is 0 when B approximate infinity.

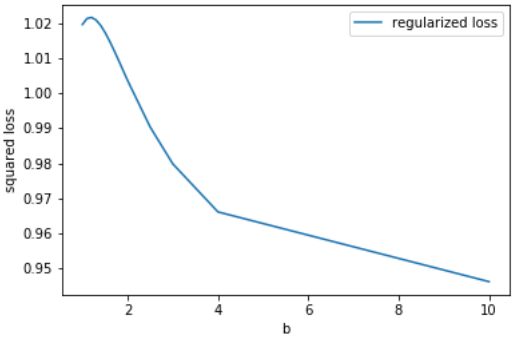
The parameter of the bias term when using 1 for the extra bias dimension can be expressed as β_0. The parameter β changes to $\frac{\beta_0}{B}$ when we use B for the extra bias dimension. The effective regularization L on the bias term $= \lambda\beta^2 = \lambda\frac{\beta_0^2}{B^2}$. When B increase, the effective regularization L on the bias term decrease. And,

$$\lim_{B\to\infty} L = \lim_{B\to\infty} \lambda\beta^2 = \lambda \lim_{B\to\infty} \frac{\beta_0^2}{B^2} = 0$$

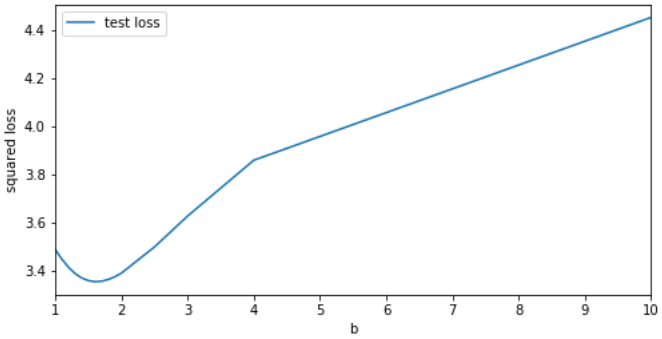
2.5.6 6

```
def regularized_grad_descent_plot_b(X_train, y_train, X_test, y_test, bs, alpha=0.01, num_iter=1000):
"""
    Plot the loss vs iteration graph using different b.
    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)
        bs - the list of b that is used to minimize loss
        alpha - step size in gradient descent
        lambda_reg - the regularization coefficient
        numIter - number of iterations to run
    """
    lambda_reg = 0.2
    num_instances, num_features = X_train.shape[0], X_train.shape[1]
    theta_hist = np.zeros((num_iter+1, num_features))  #Initialize theta_hist
    loss_hist = np.zeros(num_iter+1) #initialize loss_hist
    loss_regularize = np.zeros(len(bs))
    loss_test = np.zeros(len(bs))
    for i, b in enumerate(bs):
        X_train_b = np.copy(X_train)
        X_train_b[:,-1] *= b
        X_test_b = np.copy(X_train)
        X_test_b[:,-1] *= b
        num_instances, num_features = X_train.shape[0], X_train.shape[1]
        num_iter = int(1000*b)
        theta_hist, loss_hist = regularized_grad_descent(X_train_b, y_train, alpha=alpha/np.sqrt(b), lambda_reg = lambda_reg, num_iter=num_iter)
        theta = theta_hist[num_iter]
        loss_regularize[i] = compute_regularized_loss(theta, lambda_reg)
        loss_test[i] = compute_square_loss(X_test, y_test, theta)
    plt.figure(figsize=(8, 4))
    plt.plot(bs,loss_test,label='test loss')
    plt.xlabel("b")
    plt.ylabel("squared loss")
    plt.xlim(1,10)
    plt.legend()
    plt.savefig('2561.png')
    plt.show()
    plt.plot(bs,loss_regularize,label='regularized loss')
```

```
    plt.xlabel("b")
    plt.ylabel("squared loss")
    plt.legend()
    plt.savefig('2562.png')
    plt.show()
    return loss_regularize, loss_test

bs=[1, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2, 2.5, 3, 4, 10]
regularized_grad_descent_plot_b(X_train, y_train, X_test, y_test, bs,
 ↪ num_iter=1000)
```



The above graph show that the regularized loss decrease as b increase.

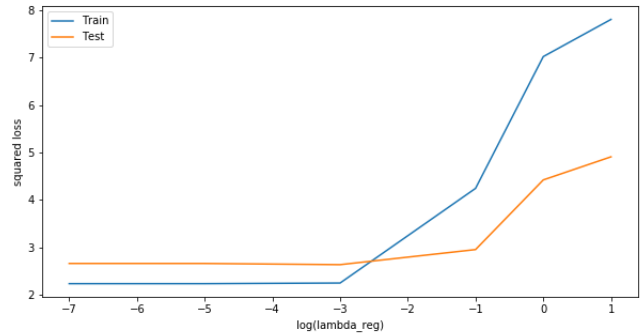


The above graph show that the test loss is minimized when b = 1.6.

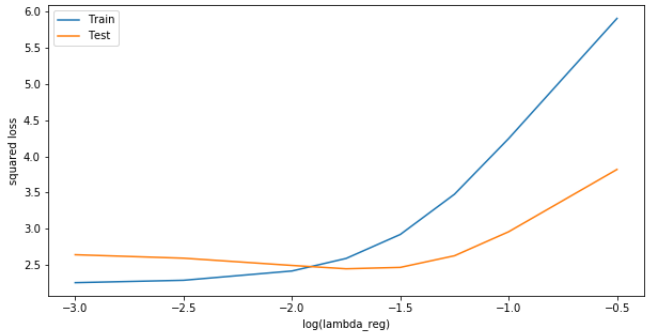
2.5.7 7

```
def regularized_grad_descent_plot_l(X_train, y_train, X_test, y_test,
 ↪ expos, alpha=0.025, num_iter=1000):
"""
    Plot the loss vs iteration graph using different lambda.
    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)
        bs - the list of b that is used to minimize loss
        alpha - step size in gradient descent
        expos- lambda_reg = 10 ** expos
        numIter - number of iterations to run

    """
    num_instances, num_features = X_train.shape[0], X_train.shape[1]
    theta_hist = np.zeros((num_iter+1, num_features))  #Initialize
     ↪ theta_hist
    loss_hist = np.zeros(num_iter+1) #initialize loss_hist
    losses_train = np.zeros(len(expos))
    losses_test = np.zeros(len(expos))
    lambda_reg=[pow(10, expo) for expo in expos]
    for i, expo in enumerate(expos):
        num_instances, num_features = X_train.shape[0],
         ↪ X_train.shape[1]
        theta_hist, loss_hist = regularized_grad_descent(X_train,
         ↪ y_train, alpha=alpha, lambda_reg = lambda_reg[i],
         ↪ num_iter=num_iter)
        theta = theta_hist[num_iter]
        losses_train[i] = compute_square_loss(X_train, y_train, theta)
        losses_test[i] = compute_square_loss(X_test, y_test, theta)
    plt.figure(figsize=(10, 5))
    plt.plot(np.log10(lambda_reg),losses_train,label='Train')
    plt.plot(np.log10(lambda_reg),losses_test,label='Test')
    plt.xlabel("log(lambda_reg)")
    plt.ylabel("squared loss")
    plt.legend()
    plt.show()
    return losses_train, losses_test
expos = [-7, -5, -3, -1, 0, 1, 2]
#expos = [-3, -2.5, -2, -1.75, -1.5, -1.25, -1, -0.5]
losses_train, losses_test = regularized_grad_descent_plot_l(X_train,
 ↪ y_train, X_test, y_test, expos, alpha=0.025, num_iter=1000)
```

The figure plots the training loss and test loss as a function of λ



As shown in the graph, the training squared loss keep increasing as λ get bigger. I also find that the test squared loss keep dropping when λ approximate -2, so I zoom in to find out what the test loss will be like.



I find that when $\lambda = 10^{-1.75}$, the test squared loss is at its minimum.

2.5.8 8

I would select the θ computed using ridge regression with $\lambda = 10^{-1.75}$. Because the test squared loss is at its minimum using the above θ, and test squared loss is the metric that determine how good the regression can generalize.

2.6 Stochastic Gradient Descent

2.6.1 1

$$f_i(\theta) = (h_\theta(x_i) - y_i)^2 + \lambda\theta_i^2$$

2.6.2 2

$$E(\nabla f_i(\theta)) = \sum_{i=1}^{m} P(x_i)\nabla f(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla (h_\theta(x_i) - y_i)^2 = \nabla J(\theta)$$

2.6.3 3

$$\theta_{i+1} = \theta_i - \eta \nabla_\theta f_i(\theta) = \theta_k - \frac{d(h_\theta(x_i) - y_i)^2 + \lambda\theta_i^2}{d\theta_i}\eta = \theta_i - 2\eta((\theta_i^T x_i - y_i)^T x_i + \lambda\theta_i)$$

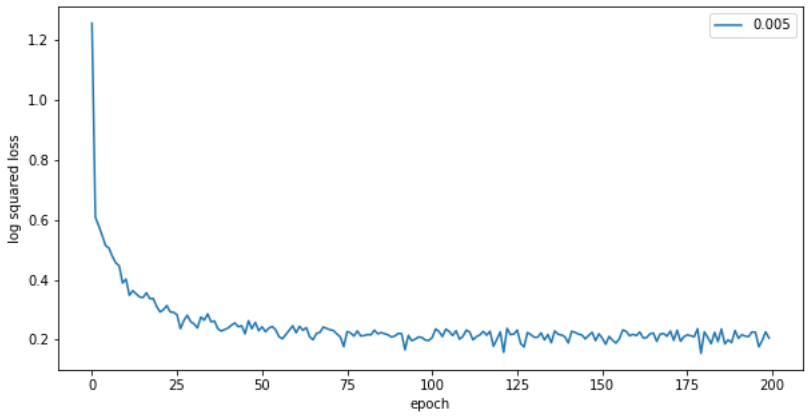
2.6.4 4

```
def stochastic_grad_descent(X_, y, alpha_=0.025, b=1.6, lambda_reg =
 ↪ pow(10,-1.75), num_iter=2000):
    """
    In this question you will implement stochastic gradient descent
 ↪ with a 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 always 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
 ↪ iteration is alpha.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t)
                if alpha == "1/t", alpha = 1/t
        lambda_reg - the regularization coefficient
        num_iter - number of epochs (i.e number of times) to go through
 ↪ the whole training set

    Returns:
        theta_hist - the history of parameter vector, 3D numpy array of
 ↪ size (num_iter, num_instances, num_features)
        loss hist - the history of regularized loss function vector, 2D
 ↪ numpy array of size(num_iter, num_instances)
    """
    X = np.copy(X_)
    X[:,-1] *= b
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) #Initialize theta
    theta_hist = np.zeros((num_iter, num_instances, num_features))
     ↪ #Initialize theta_hist
    loss_hist = np.zeros((num_iter, num_instances)) #Initialize
     ↪ loss_hist
    losses = np.zeros(num_iter)
    temp_index = 0
    start = 0
    if not isinstance(alpha_,float):
        start = 100
    for i in range(start,num_iter):
        if not isinstance(alpha_,float):
            if (alpha_ == "1/t"):
                alpha = 1/i
```

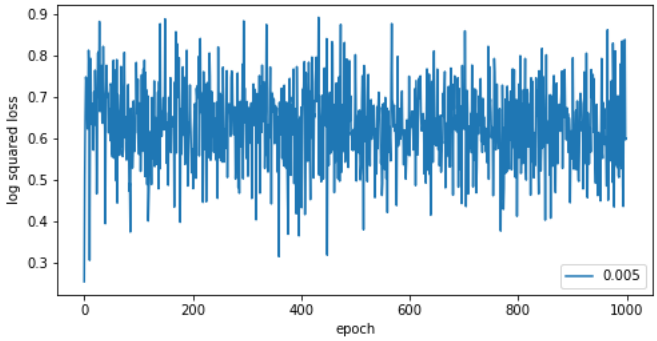
```
            elif (alpha_ == "1/sqrt(t)"):
                alpha =  1/np.sqrt(i)
            elif (alpha_ == "mode3"):
                alpha = 0.01
                alpha =  alpha/(1+alpha*lambda_reg*i)
            else:
                print ("erro")
                return None, None
        else:
            alpha = alpha_
        index = np.array(range(num_instances))
        np.random.shuffle(index)
        for k, j in enumerate(index):
            gradient = compute_regularized_square_loss_gradient(X[j,
             ↪  :].reshape(1,X[j, :].shape[0]), y[j], theta,
             ↪  lambda_reg)
            theta = theta - alpha * gradient
            theta_hist[i-start, j] = theta
            loss = compute_square_loss(X[j, :].reshape(1,X[j,
             ↪  :].shape[0]), y[j], theta, lambda_reg)
            if loss > 999999999999:
                print( '{} overflow'.format(alpha_))
                return None, None
            loss_hist[i-start, j] = loss
    return theta_hist, loss_hist
```

2.6.5 5

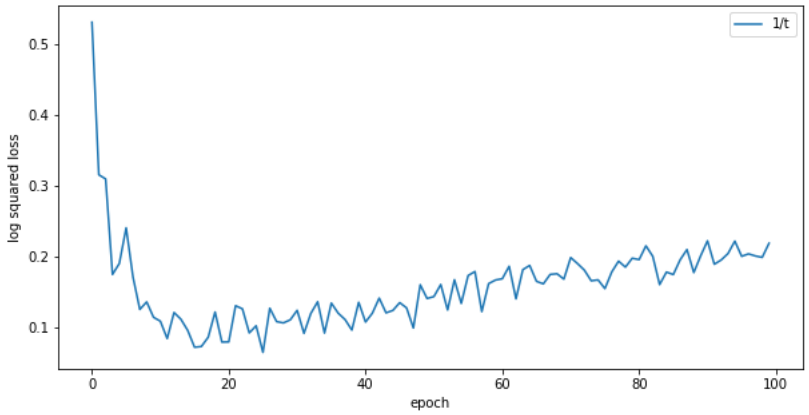
I try SGD using b=1.25, $\lambda = 10^{-1.75}$ with fixed step sizes. It turns out that $alpha = 0.05$ don't converge whereas that $alpha = 0.005$ converge. The graph for SGD with step size 0.005 is as follows. The Y axis is the log average squared loss for all the instances. The X axis is the epoch.



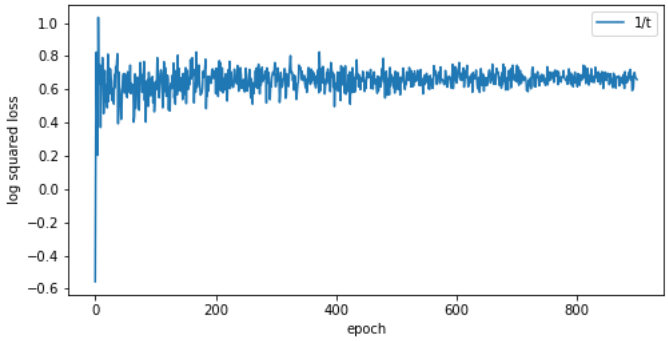
The following graph shows loss for the last instance of the 100 reshuffled instances vs epoch with fixed step size = 0.05. The instance loss is much messier than the average loss.



For decreased step sizes, I try both $1/t$ and $1/\sqrt{(t)}$ with first 100 steps omitted. STD don't converge with $1/\sqrt{(t)}$. The graph for SGD with step size $1/t$ is as follows.

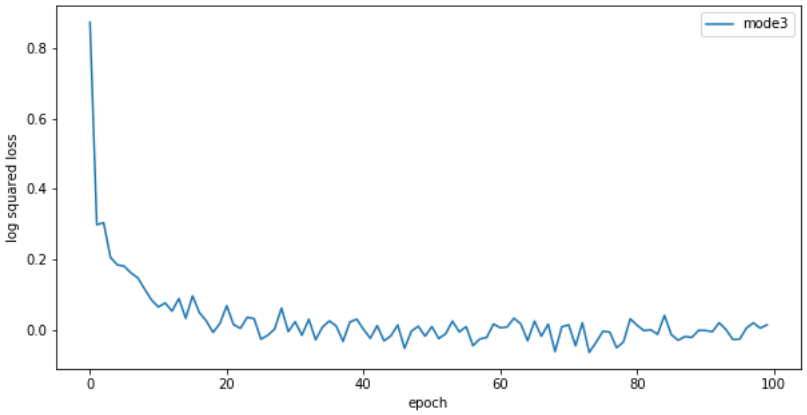


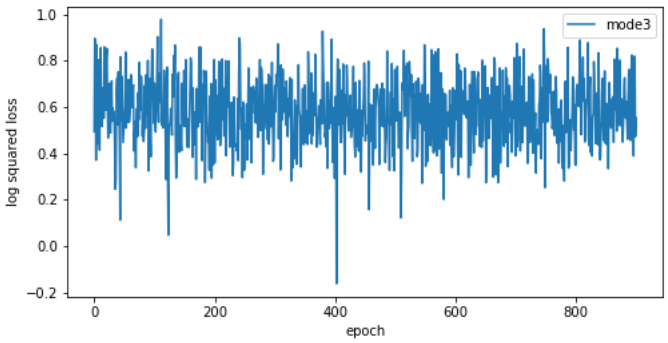
We can see that the SGD with decreasing step sizes converge faster than the one with fixed step sizes. Also, the former one also achieve better results than the later one.
The following graph shows loss for the last instance of the 100 reshuffled instances vs epoch with decreased step size $\frac{1}{t}$. We can see that the change of loss is getting smaller while the step size is getting smaller.



2.6.6 6

For step sizes of the form $\frac{\eta_0}{1+\eta_0 \lambda t}$, the graph for SGD is as follows.
We can see that SGD with this from achieve the best result among all three. However, its converge speed is not as fast as the one with step size $1/t$.





The above graph shows loss for the last instance of the 100 reshuffled instances vs epoch with decreased step size $\frac{\eta_0}{1+\eta_0 \lambda t}$.

3 Risk Minimization

3.1 Square Loss

3.1.1 1

$$\mathbb{E}\left(a-y\right)^2 = Var(a-y) + \left(E(a-y)\right)^2 = Var(y) + \left(E(a-y)\right)^2 = Var(y) + \left(a - E(y)\right)^2$$

When $a = E(y)$, the expected squared loss is at its minimum. The minimum squared loss is the variance of the distribution.

3.1.2 2a

$$f^*(x) = E(y|x)$$

the expected loss is $Var(y|x)$

3.1.3 2b

$$\mathbb{E}\left[\mathbb{E}\left[(f^*(x)-y)^2 \mid x\right]\right] = \mathbb{E}\left[(f^*(x)-y)^2\right] \quad (1)$$

$$\mathbb{E}\left[\mathbb{E}\left[(f(x)-y)^2 \mid x\right]\right] = \mathbb{E}\left[(f(x)-y)^2\right] \quad (2)$$

According to 3.1.2a, (2) is bigger than (1) when $f(x)! = f^*(x)$, therefore

$$\mathbb{E}\left[(f^*(x)-y)^2 \mid x\right] \leq \mathbb{E}\left[(f(x)-y)^2 \mid x\right]$$

3.2 [Optional] Median Loss

We aim to find $f^*(x) = \arg\min_f(x) E((|y - f(x)|) \mid x)$
Let $a = f^*(x)$

$$\frac{dE((|y-a|) \mid x)}{da}$$
$$= E(1(y - a > 0) * (-1) + 1(a - y < 0) * 1)$$

Let $E(1(y - a > 0) * (-1) + 1(a - y < 0) * 1) = 0$ we obtain

$$P(y - a > 0) = P(a - y < 0)$$

$$\Rightarrow P(y \geq a) \geq \frac{1}{2}$$

$$P(y \leq a) \geq \frac{1}{2}$$

$\Rightarrow a$, that is $f^*(x)$ is the median of conditional distribution of y given x