ML hw2

March 10, 2020

1 Gradescope

Author: Yibo Liu (yl6769)

2 Computing Risk

2.1 Expectations

1(a) $E[||\vec{x}||_2^2] = E[x_1^2 + ... + x_n^2] = nE[x_i^2] = n((-2)^2 + (-1)^2 + 0^2 + 1^2 + 2^2)/5 = 2n$
1(b) $E[||\vec{x}||_\infty] = E[max_i|x_i|] = (-2)/5 + 2/5 = 4/5$
1(c) For the elements in $\Sigma_{\vec{x}}$, $\Sigma_{ii} = Var[x_i] = E[x_i^2] - E[x_i]^2 = 2, \Sigma_{ij} = 0 (i \neq j)$

2.2 Bayes risk

2(a) We have $E[a] = a, Var[a] = 0$
$E[(a-y)^2] = E[a^2 + y^2 - 2ay] = E[a^2] + E[y^2] - 2E[ay] = Var(a) + E[a]^2 + Var(y) + E[y]^2 - 2aE[y] = Var[y] + (a - E[y])^2$
Therefore, $a^* = E[y]$, the Bayes risk is $Var[y]$

2.3 Bayes decision function

2(b)(i)

We have $E[a|x] = a, Var[a|x] = 0$, because a is a deterministic function of x.
$E[(a-y)^2|x] = E[a^2 + y^2 - 2ay|x] = E[a^2|x] + E[y^2|x] - 2E[ay|x] = Var(a|x) + E[a|x]^2 + Var(y|x) + E[y|x]^2 - 2aE[y|x] = Var[y|x] + (a - E[y|x])^2$
Therefore, $a^* = E[y|x]$, the Bayes risk is $Var[y|x]$

2(b)(ii)

$E[(f^*(x) - y)^2] = E[E[(f^*(x) - y)^2|x]] \leq E[E[(f(x) - y)^2|x]] = E[(f(x) - y)^2]$
The first and the second equality uses the law of iterated expectations. The inequality uses the given fact that $E[(f^*(x) - y)^2|x] \leq E[(f(x) - y)^2|x]$, Since the expectations are scaler, so the inequality of the expectations of these expectations is also true.

3 Linear Regression

3.1 Feature normalization

1(a)

[5]:
```
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

[6]:
```
########################################
### Feature normalization
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
    """
    train_min = np.amin(train, axis=0)
    train_max = np.amax(train, axis=0)
    train_range = train_max - train_min

    del_li = []
    for i in range(len(train_range)):
        if train_range[i] == 0:
            del_li.append(i)

    train = np.delete(train, del_li, axis=1)
    test = np.delete(test, del_li, axis=1)
    train_range = np.delete(train_range, del_li)
    train_min = np.delete(train_min, del_li)

    train_normalized = (train - train_min) / train_range
    test_normalized = (test - train_min) / train_range

    return train_normalized, test_normalized
```

[7]:
```
#Loading the dataset
print('loading the dataset')

df = pd.read_csv('ridge_regression_dataset.csv', delimiter=',')
X = df.values[:,:-1]
y = df.values[:,-1]

print('Split into Train and Test')
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =100,␣
 ↪random_state=10)

print("Scaling all to [0, 1]")
X_train, X_test = feature_normalization(X_train, X_test)
X_train = np.hstack((X_train, np.ones((X_train.shape[0], 1))))  # Add bias term␣
 ↪(B=1)
X_test = np.hstack((X_test, np.ones((X_test.shape[0], 1))))  # Add bias term
```

```
loading the dataset
Split into Train and Test
Scaling all to [0, 1]
```

3.2 Objective function

2(a) $J = \frac{1}{m}\sum_{i=1}^{m}(\theta^T x_i - y_i)^2 = \frac{1}{m}|X\theta - Y|^2 = \frac{1}{m}(X\theta - Y)^T(X\theta - Y)$

3.3 Gradient of objective function

2(b) $\nabla J = \frac{2}{m}X^T(X\theta - Y)$

3.4 Using first order approximation

2(c) $J(\theta + \eta h) \simeq J(\theta) + \eta h^T \nabla J(\theta)$

3.5 Update expression

2(d) $\theta' = \theta - \eta \nabla J(\theta)$

3.6 Compute square loss

2(e)

[8]:
```
########################################
### The square loss function
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
    m = X.shape[0]
    diff = np.dot(X, theta) - y
    loss = 1/(2*m) * np.dot(diff, diff)
    return loss
```

3.7 Compute square loss gradient

2(f)

[9]:
```
########################################
### The gradient of the square loss function
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
    m = X.shape[0]
    diff = np.dot(X, theta) - y
    grad = (1/m) * np.dot(diff, X)
    return grad
```

3.8 Gradient checker

3(a)

[10]:
```
#####################################
### Gradient checker
#Getting the gradient calculation correct is often the trickiest part
#of any gradient-based optimization algorithm. Fortunately, it's very
#easy to check that the gradient calculation is correct using the
#definition of gradient.
#See http://ufldl.stanford.edu/wiki/index.php/
 ↪Gradient_checking_and_advanced_optimization
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
    """Implement Gradient Checker
    Check that the function compute_square_loss_gradient returns the
    correct gradient for the given X, y, and theta.

    Let d be the number of features. Here we numerically estimate the
    gradient by approximating the directional derivative in each of
    the d coordinate directions:
    (e_1 = (1,0,0,...,0), e_2 = (0,1,0,...,0), ..., e_d = (0,...,0,1))

    The approximation for the directional derivative of J at the point
    theta in the direction e_i is given by:
    ( J(theta + epsilon * e_i) - J(theta - epsilon * e_i) ) / (2*epsilon).

    We then look at the Euclidean distance between the gradient
    computed using this approximation and the gradient computed by
    compute_square_loss_gradient(X, y, theta).  If the Euclidean
    distance exceeds tolerance, we say the gradient is incorrect.

    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)
        epsilon - the epsilon used in approximation
        tolerance - the tolerance error

    Return:
        A boolean value indicating whether the gradient is correct or not
    """
    true_gradient = compute_square_loss_gradient(X, y, theta) #The true gradient
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we approximate
    #TODO
    for i in range(num_features):
        e_i = np.zeros(num_features)
        e_i[i] = 1
        approx_grad[i] = 1/(2*epsilon) *␣
 ↪(compute_square_loss(X,y,(theta+epsilon*e_i)) -
```

```
                                        compute_square_loss(X,y,(theta-epsilon*e_i)))
    return np.linalg.norm(true_gradient - approx_grad) < tolerance
```

[11]:
```
########################################
### Generic gradient checker
def generic_gradient_checker(X, y, theta, objective_func, gradient_func,␣
 ↪epsilon=0.01, tolerance=1e-4):
    """
    The functions takes objective_func and gradient_func as parameters.
    And check whether gradient_func(X, y, theta) returned the true
    gradient for objective_func(X, y, theta).
    Eg: In LSR, the objective_func = compute_square_loss, and gradient_func =␣
 ↪compute_square_loss_gradient
    """
    #TODO
    true_gradient = gradient_func(X, y, theta) #The true gradient
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we approximate
    #TODO
    for i in range(num_features):
        e_i = np.zeros(num_features)
        e_i[i] = 1
        approx_grad[i] = 1/(2*epsilon) *␣
 ↪(objective_func(X,y,(theta+epsilon*e_i)) -
                                        objective_func(X,y,(theta-epsilon*e_i)))
#     print(true_gradient, approx_grad)
    return np.linalg.norm(true_gradient - approx_grad) < tolerance
```

3.9 Batch gradient descent

4(a) (Taking all examples as a batch)

[12]:
```
########################################
### 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
    grad_err = False
    theta_hist[0] = theta
    for i in range(num_step):
        loss = compute_square_loss(X, y, theta)
        loss_hist[i] = loss
        grad = compute_square_loss_gradient(X, y, theta)
        if grad_check and grad_checker(X,y,theta) == False:
            print("alpha=",alpha,"grad error at step", i)
            grad_err = True
            break
        theta = theta - alpha * grad
        theta_hist[i+1] = theta
    loss_hist[i+1] = compute_square_loss(X, y, theta)
    return theta_hist, loss_hist, grad_err
```

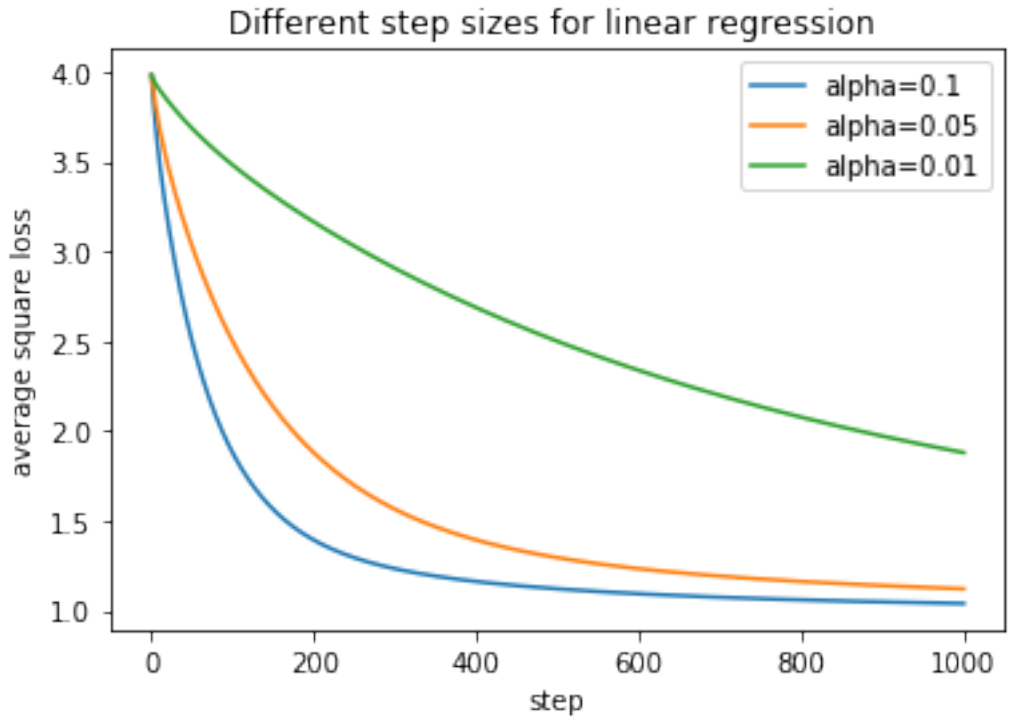
3.10 Experiment on step size

4(b)

[18]:
```
li = [0.5, 0.4, 0.3, 0.2, 0.1, 0.05, 0.01]
for alpha in li:
    theta_hist, loss_hist, grad_err = batch_grad_descent(X_train, y_train,␣
 ↪alpha=alpha, grad_check=True)
    if grad_err == False:
        plt.plot(range(len(theta_hist)),loss_hist,label="alpha="+str(alpha))
plt.title('Different step sizes for linear regression ')
plt.ylabel('average square loss')
plt.xlabel('step')
plt.legend(loc = 1)
plt.show()
```

```
alpha= 0.5 grad error at step 6
alpha= 0.4 grad error at step 7
```

```
alpha= 0.3 grad error at step 9
alpha= 0.2 grad error at step 12
```



Conclusion: When alpha(step size)=0.1, 0.05, 0.01, the average square loss converges, and it converges faster when alpha is larger. However, when alpha grows larger, e.g. alpha=0.2,0.3,0.4,0.5, the average square loss does not converge.

4 Ridge Regression

4.1 Vector gradient of J

$\nabla J(\theta) = \frac{2}{m} X^T (X\theta - Y) + 2\lambda\theta^T$
Updating θ: $\theta' = \theta - \eta \nabla J(\theta)$

4.2 Compute regularized square loss gradient

[13]:
```
########################################
### The gradient of regularized batch gradient descent
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    """
    Compute the gradient of L2-regularized average 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)
    """
    #TODO
    grad = compute_square_loss_gradient(X, y, theta) + lambda_reg * theta
    return grad
```

4.3 Regularized gradient descent

[14]:
```
########################################
### Regularized batch gradient descent
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2, num_step=1000,␣
↪grad_check=False):
    """
    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
        num_step - number of steps to run

    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+1) is theta_hist[-1]
        loss hist - the history of average square loss function without the␣
↪regularization term, 1D numpy array.
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.zeros(num_features) #Initialize theta
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    #TODO
    grad_err = False
    theta_hist[0] = theta
    for i in range(num_step):
        loss = compute_square_loss(X, y, theta)
        loss_hist[i] = loss
```

```
        grad = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)
        if grad_check and grad_checker(X,y,theta) == False:
            print("alpha=",alpha,"grad error at step", i)
            grad_err = True
            break
        theta = theta - alpha * grad
        theta_hist[i+1] = theta
    loss_hist[i+1] = compute_square_loss(X, y, theta)
    return theta_hist, loss_hist, grad_err
```

4.4 Explain bias in optimization

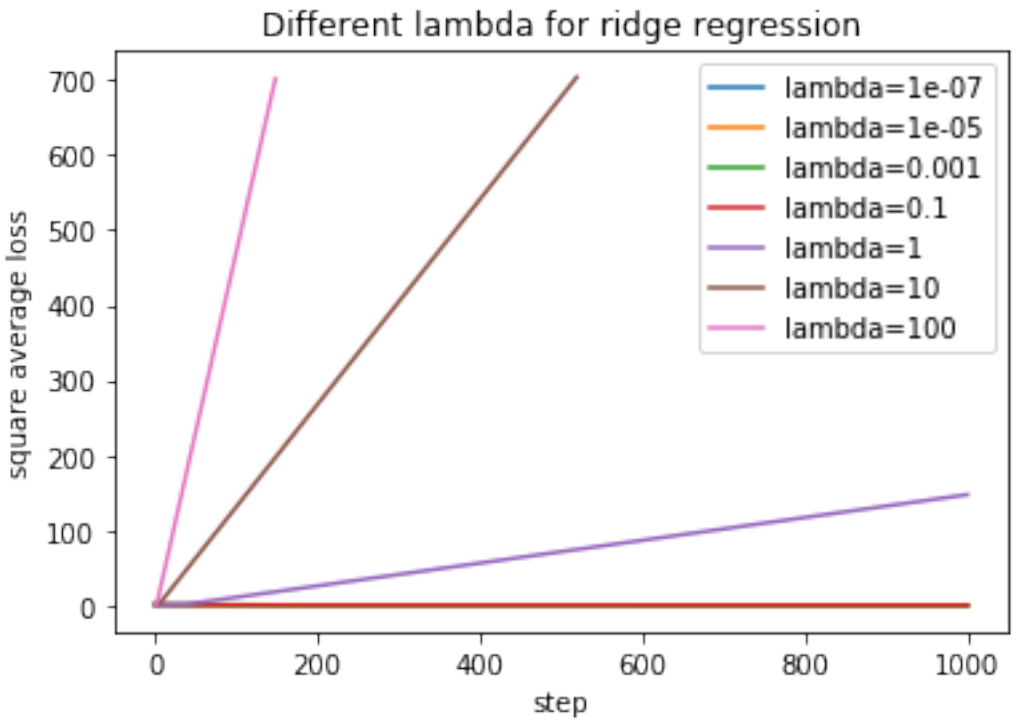
The bias term is the product of B and the last column in theta. When applying a regularization, if B is very large, the adjustment on theta would be small. Therefore, the effectiveness of regularization on bias term is decreased. We could set B approching infnity to make the regularization negligible.

4.5 Find optimal parameter

Find the lambda that minimize the average square loss (without regularization part) on the test set. First we fix B=1, set step size (alpha) to 0.1 (which is the optimal alpha from the previous experiments) and try a range of lambda.

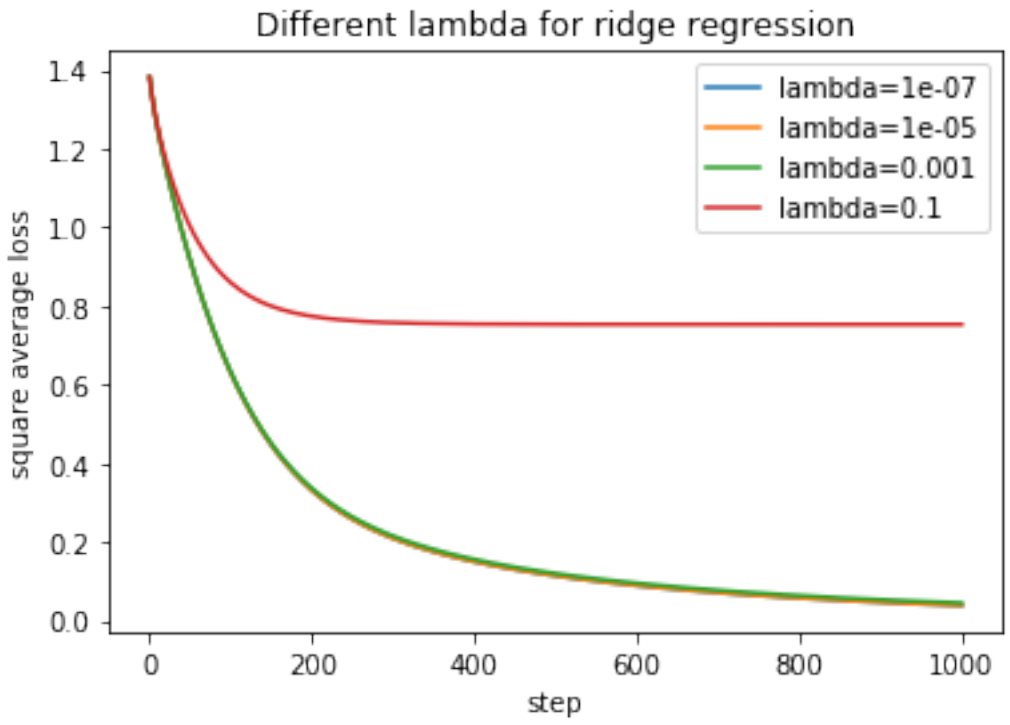
[27]:
```
def plt_lambda_reg(li):
    a = 0.1
    # TODO: choose alpha from backtracking line search
    train_loss = []
    test_loss = []
    for l in li:
        theta_hist, loss_hist, grad_err =␣
↪regularized_grad_descent(X_train,y_train, alpha=a, lambda_reg=l, num_step=1000)
        plt.plot(range(len(loss_hist)),np.log(loss_hist),label="lambda="+str(l))
        train_loss.append(loss_hist[-1])
    plt.title('Different lambda for ridge regression')
    plt.xlabel("step")
    plt.ylabel("square average loss")
    plt.legend(loc = 1)
    plt.show()
```

[28]:
```
plt_lambda_reg([1e-7,1e-5,1e-3,1e-1,1, 10,100])
```



When lambda=1, 10 or 100, the loss diverges.

[29]:
```
plt_lambda_reg([1e-7,1e-5,1e-3,1e-1])
```

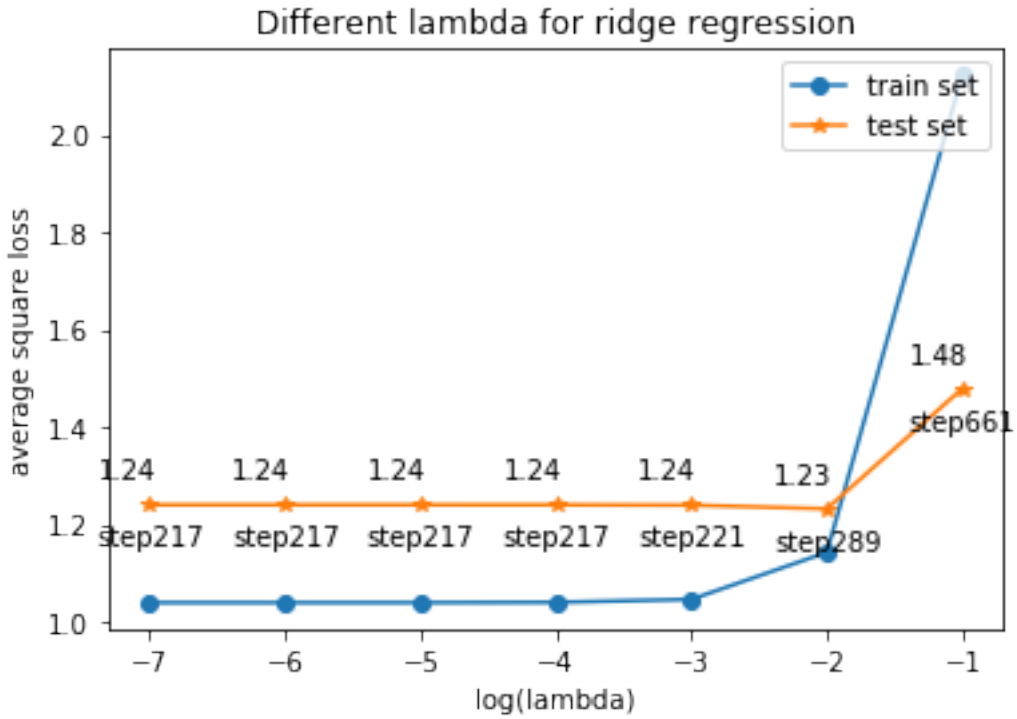


When lambda=0.1,0.001,1e-5,1e-7, the loss converges.

In order to find the lambda with the minimal loss on test set, we compare *the minimal loss among all steps* for different lambda.

[66]:
```
a = 0.1
# TODO: choose alpha from backtracking line search
li = [1e-7,1e-6,1e-5,1e-4,1e-3,1e-2,1e-1]
train_loss = []
test_loss = []
steps = []
for l in li:
    # train loss
    theta_hist, loss_hist, grad_err = regularized_grad_descent(X_train,y_train,␣
 ↪alpha=a, lambda_reg=l, num_step=1000)
    train_loss.append(loss_hist[-1])
    # test loss
    test_loss_hist= [compute_square_loss(X_test,y_test,theta) for theta in␣
 ↪theta_hist]
    # find the minimum test loss with theta from all training steps
#     test_loss.append(test_loss_hist[-1])
    i = np.argsort(test_loss_hist)[0]
    test_loss.append(test_loss_hist[i])
    steps.append(i)
```

```
plt.plot(np.log10(li),train_loss, marker='o',label='train set')
plt.plot(np.log10(li),test_loss, marker='*',label='test set')
i = 0
for xy in zip(np.log10(li), test_loss):
    plt.annotate("step%s" % steps[i], xy=xy, xytext=(-20, -15),␣
 ↪textcoords='offset points')
    plt.annotate("%s" % test_loss[i].round(2), xy=xy, xytext=(-20, +10),␣
 ↪textcoords='offset points')
    i += 1
plt.title('Different lambda for ridge regression')
plt.xlabel("log(lambda)")
plt.ylabel("average square loss")
plt.legend(loc = 1)
plt.show()
```



We choose labmda=0.01 as the optimal lambda, since test loss reaches the minimum.

4.6 What theta to select

In practice, choose the theta when using lambda=0.01 and step=289, because it achieves the minimum loss on test set.

[22]:
```
theta_hist, loss_hist, _ = regularized_grad_descent(X_train,y_train, alpha=0.1,␣
↪lambda_reg=0.01, num_step=1000)
print('the optimal theta:\n', theta_hist[289].round(4))
```

```
the optimal theta:
 [-1.1473  0.5415  1.4387  2.0905 -1.5139 -0.8571 -0.7966 -0.7966  0.6355
  1.4345  2.1797 -0.2793 -1.5616 -3.6506  1.4153  2.2064  1.3725  0.3455
 -0.0853 -0.0853 -0.0853 -0.0118 -0.0118 -0.0118  0.0179  0.0179  0.0179
  0.0323  0.0323  0.0323  0.0406  0.0406  0.0406 -0.037  -0.037  -0.037
  0.1025  0.1025  0.1025  0.0849  0.0849  0.0849  0.0769  0.0769  0.0769
  0.0725  0.0725  0.0725 -1.3025]
```

5 Stochastic Gradient Descent

5.1 Objective function equivalence

$f_i(\theta) = (h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta$
$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta) = \frac{1}{m}\sum_{i=1}^{m}((h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta$

5.2 Prove unbiased estimator

Left of the equation: $E[\nabla f_i(\theta)] = \sum_{i=1}^{m} P(i)\nabla f_i(\theta) = \sum_{i=1}^{m} \frac{1}{m}\nabla f_i(\theta)$
Right of the equation: $\nabla J(\theta) = \nabla(\frac{1}{m}\sum_{i=1}^{m} f_i(\theta)) = \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(\theta)$
Therefore, $E[\nabla f_i(\theta)] = \nabla J(\theta)$

5.3 Update rule

$\theta' = \theta - stepsize * \nabla f_i(\theta)$ for every sample x_i

5.4 Implement SGD

[8]:
```
########################################
### Stochastic gradient descent
import random
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2, num_epoch=1000,␣
↪c=0.1):
    """
    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.
        lambda_reg - the regularization coefficient
        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
    for i in range(num_epoch):
        # shuffle
        idx = np.arange(num_instances)
        random.shuffle(idx)
        X = X[idx]
        y = y[idx]
        loss = 0
        for j in range(num_instances):
            if alpha == '1/sqrt(t)':
                alpha = c/np.sqrt((i*num_epoch+j+1))
            elif alpha == '1/t':
                alpha = c/(i*num_epoch+j+1)

            theta_hist[i][j] = theta
            diff = np.dot(X[j],theta) - y[j]
            loss = np.dot(diff,diff) + lambda_reg * np.dot(theta,theta)
            loss_hist[i][j] = loss

            grad = 2*(np.dot(diff,X[j])+lambda_reg*theta)
            theta = theta - alpha * grad
#         if i%100==0:
#             print('epoch=',i,'loss=',np.mean(loss_hist[i],axis=0))
    return theta_hist, loss_hist
```

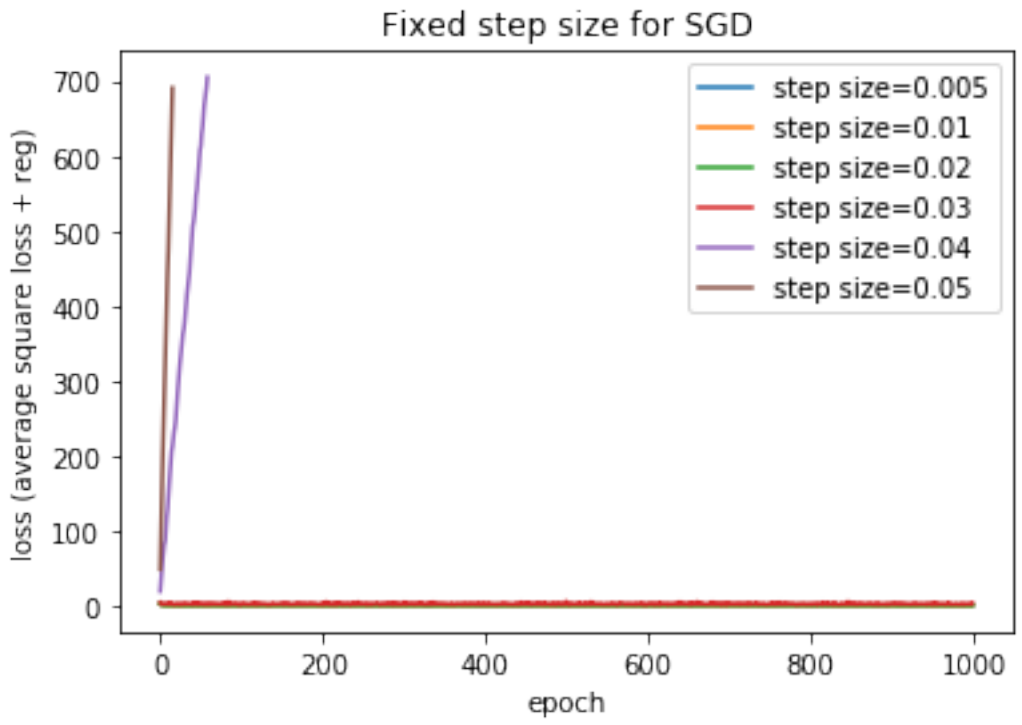
5.5 Find optimal parameters

(1) Fixed step size

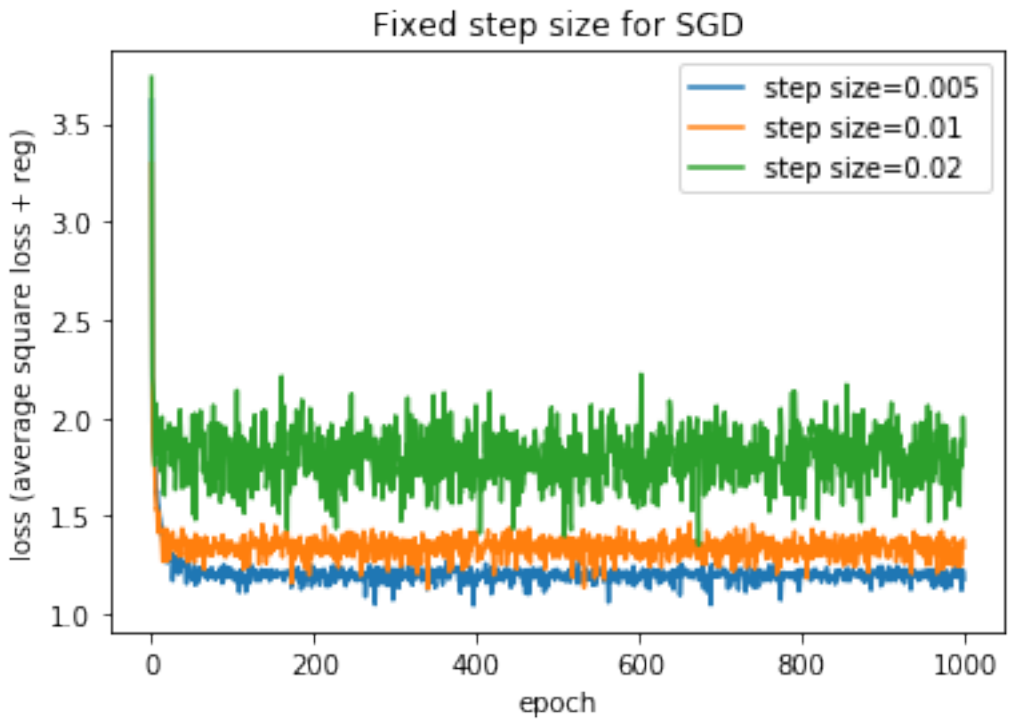
[77]:
```
def plt_fix_sgd(li):
    for alpha in li:
        theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train,␣
↪alpha=alpha)
        avg_loss_hist = np.mean(loss_hist, axis=1)
        plt.plot(range(len(avg_loss_hist)),np.log(avg_loss_hist),label="step␣
↪size="+str(alpha))
    plt.title('Fixed step size for SGD')
    plt.xlabel("epoch")
    plt.ylabel("loss (average square loss + reg)")
    plt.legend(loc = 1)
    plt.show()
```

[78]:
```
plt_fix_sgd([0.005,0.01,0.02,0.03,0.04,0.05])
```

```
/opt/conda/envs/dsga-1003/lib/python3.7/site-packages/ipykernel_launcher.py:48:
RuntimeWarning: overflow encountered in multiply
```



[79]:
```
plt_fix_sgd([0.005,0.01,0.02])
```

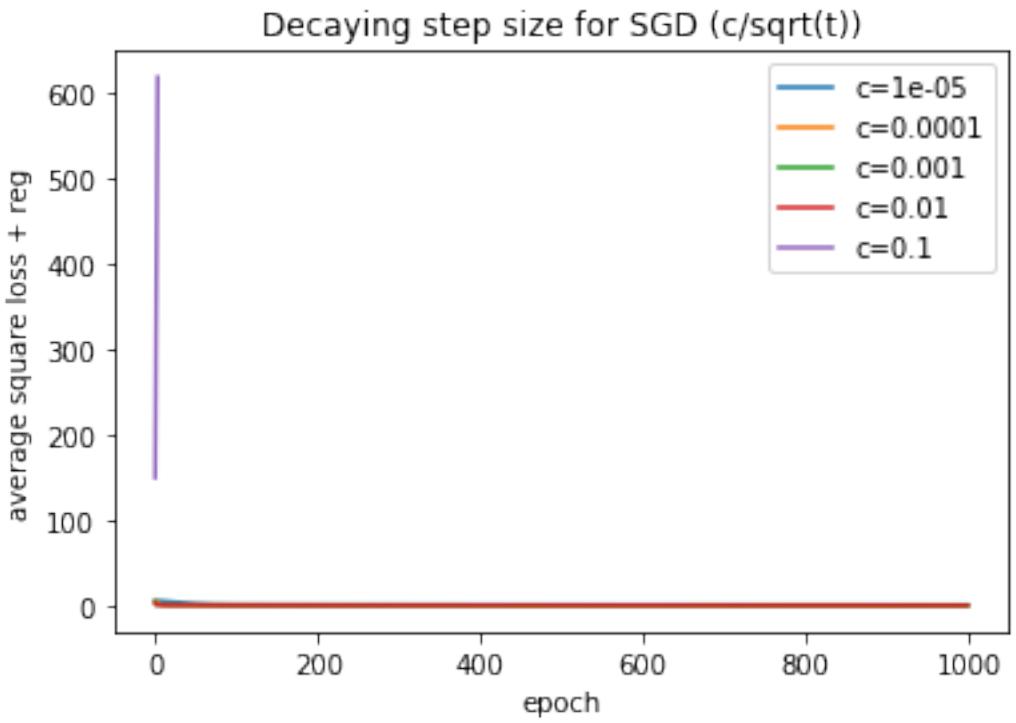


(2) Decaying step size using different schedules

[29]:
```
def plt_decay_sgd(li, alpha):
    for c in li:
        theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train,␣
↪alpha=alpha, c=c)
        avg_loss_hist = np.mean(loss_hist, axis=1)
        plt.plot(range(len(avg_loss_hist)),np.
↪log(avg_loss_hist),label="c="+str(c))
    plt.title('Decaying step size for SGD (c'+ alpha[1:] + ')')
    plt.xlabel("epoch")
    plt.ylabel("average square loss + reg")
    plt.legend(loc = 1)
    plt.show()
```

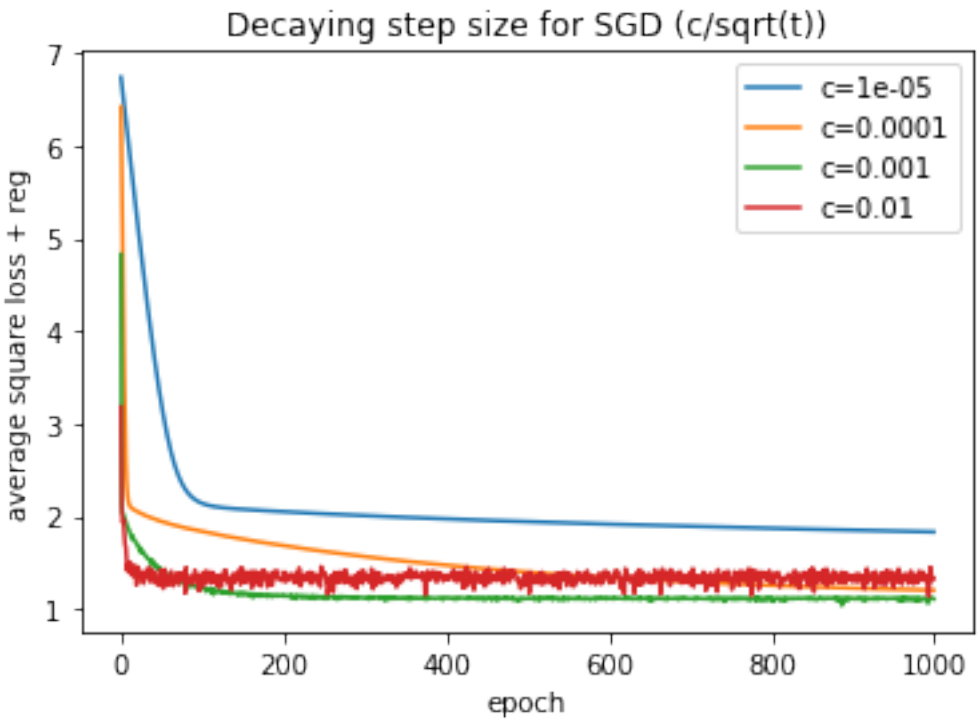
[30]:
```
plt_decay_sgd([1e-5, 1e-4, 1e-3, 1e-2, 0.1], '1/sqrt(t)')
```

```
/opt/conda/envs/dsga-1003/lib/python3.7/site-packages/ipykernel_launcher.py:51:
RuntimeWarning: overflow encountered in multiply
```



When the initial step size c=0.1, it is too aggressive that the loss function does not converge.

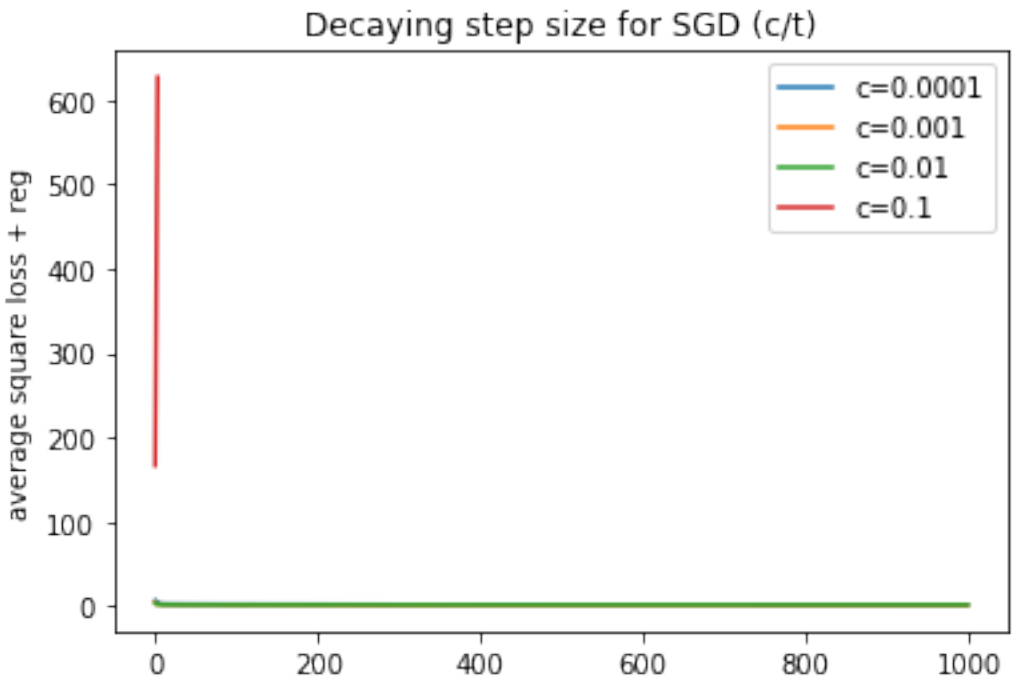
[32]:
```
plt_decay_sgd([1e-5, 1e-4, 1e-3, 1e-2], '1/sqrt(t)')
```



The when c<0.001, the loss function converges too slow. When c>0.001, it does not reach the minimum loss. So the best configuration is c=0.001 for the schema *step size = c/sqrt(t)*.

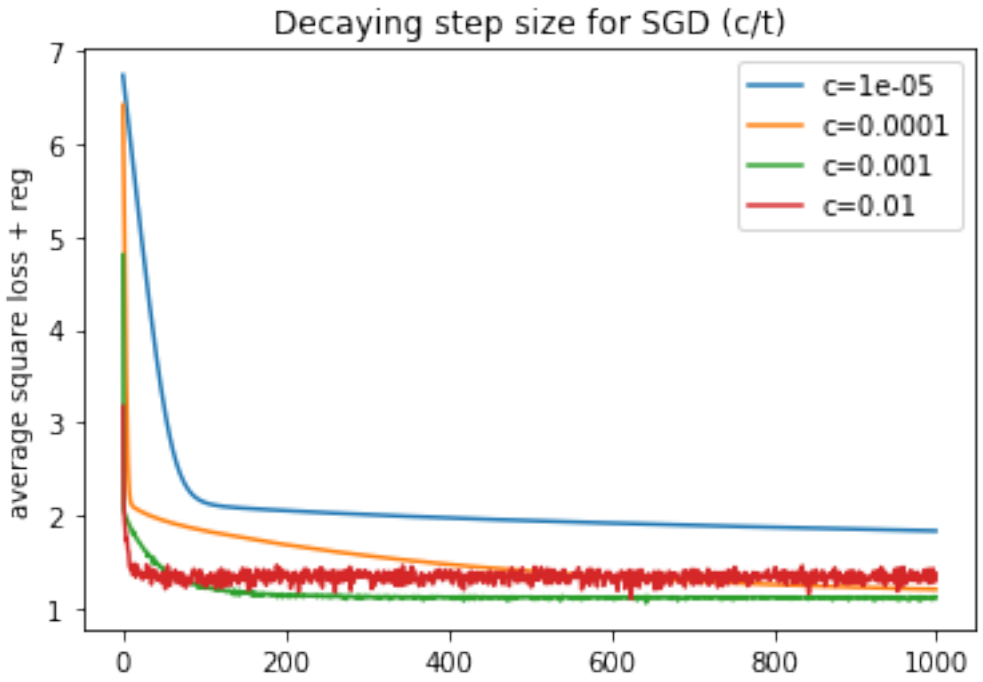
[103]:
```
plt_decay_sgd([0.0001, 0.001, 0.01, 0.1], '1/t')
```

```
/opt/conda/envs/dsga-1003/lib/python3.7/site-packages/ipykernel_launcher.py:48:
RuntimeWarning: overflow encountered in multiply
```



When the initial step size c=0.1, it is too aggressive that the loss function does not converge.

[100]:
```
plt_decay_sgd([1e-5, 1e-4, 1e-3, 1e-2], '1/t')
```

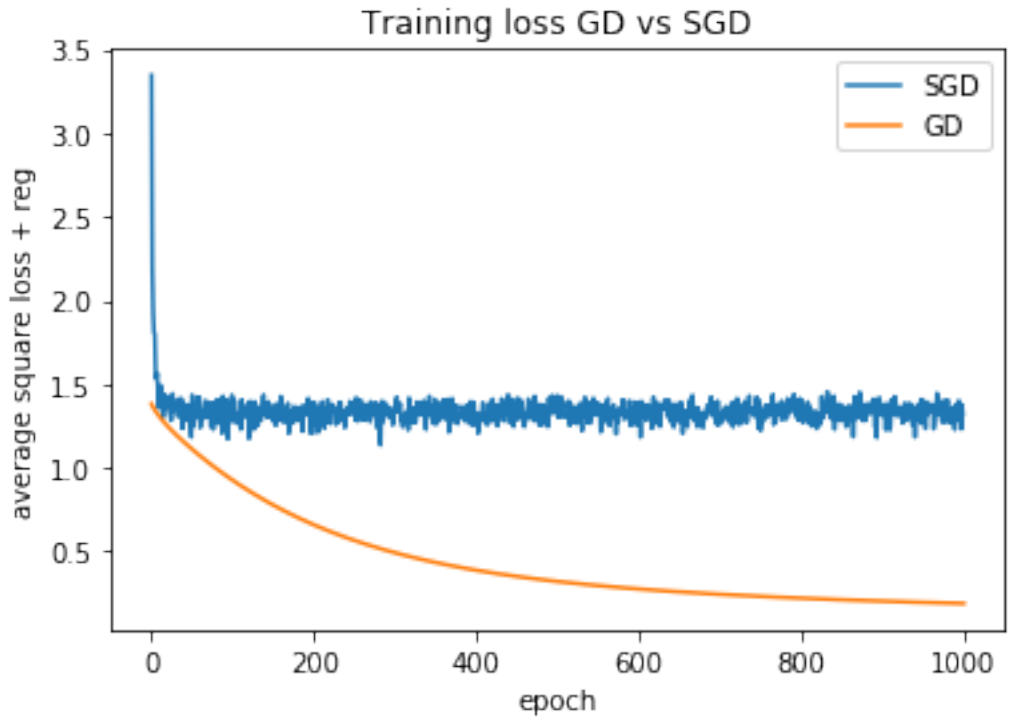


The when c<0.001, the loss function converges too slow. When c>0.001, it does not reach the minimum loss. So the best configuration is c=0.001 for the schema *step size = c/t*.

(3) Compare GD with SGD

[94]:
```
import time
start_time = time.time()
theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train)
print('SGD training time for 1000 epoch',time.time()-start_time)
avg_loss_hist = np.mean(loss_hist, axis=1)
plt.plot(range(len(avg_loss_hist)),np.log(avg_loss_hist),label='SGD')

start_time = time.time()
theta_hist, loss_hist, _ = regularized_grad_descent(X_train, y_train)
print('GD training time for 1000 epoch',time.time()-start_time)
plt.plot(range(len(loss_hist)),np.log(loss_hist),label='GD')

plt.title('Training loss GD vs SGD')
plt.xlabel("epoch")
plt.ylabel("average square loss + reg")
plt.legend(loc = 1)
plt.show()
```

```
SGD training time for 1000 epoch 1.841592788696289
GD training time for 1000 epoch 0.02275848388671875
```



It shows that SDG converges slower than GD once we get close to the minimizer.

5.6 Adaptive step size

(1) average theta

[21]:
```
def compute_sgd_loss(X,y,theta, lambda_reg):
    m = X.shape[0]
    diff = np.dot(X, theta) - y
    loss = 1/m * np.dot(diff, diff) + lambda_reg * np.dot(theta,theta)
    return loss
lambda_reg = 0.01
theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train,␣
 ↪lambda_reg=lambda_reg)
test_loss_last = compute_sgd_loss(X_test,y_test,theta_hist[-1,-1,:
 ↪],lambda_reg=lambda_reg)
test_loss_avg = compute_sgd_loss(X_test,y_test,np.
 ↪mean(theta_hist,axis=(0,1)),lambda_reg=lambda_reg)
print('test loss with last step theta:', test_loss_last.round(4), '\ntest loss␣
 ↪with average theta:', test_loss_avg.round(4))
```

```
test loss with last step theta: 4.0259
test loss with average theta: 3.285
```

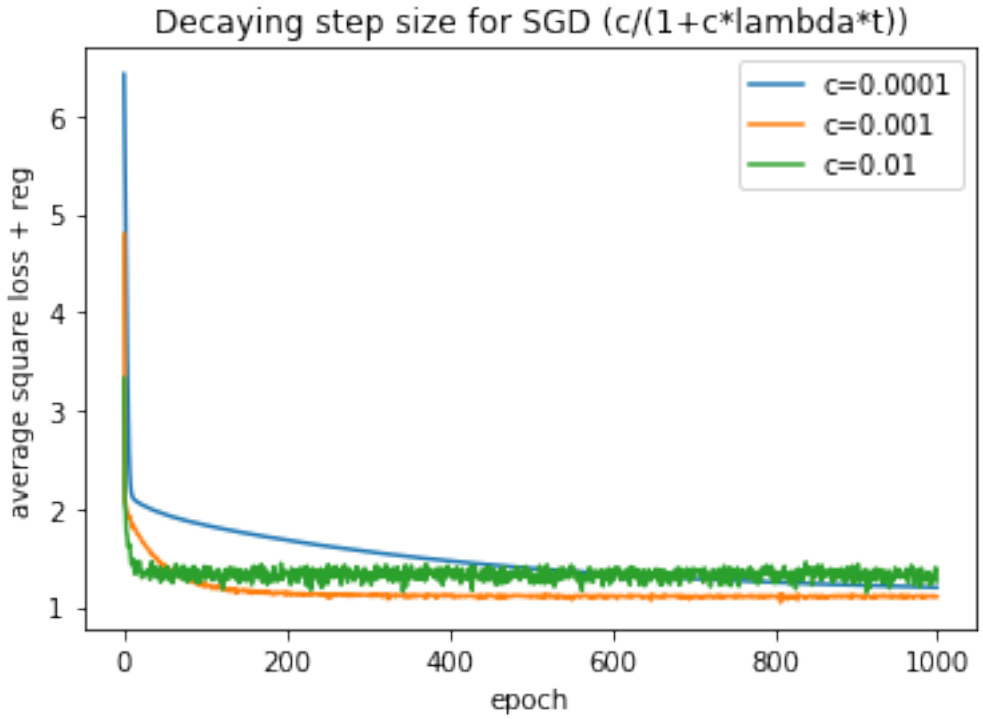
(2) The new stepsize rule: $stepsize = \frac{c}{1+c\lambda t}$

[23]:
```
########################################
### Stochastic gradient descent
import random
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2, num_epoch=1000,␣
 ↪c=0.1):
    """
    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.
        lambda_reg - the regularization coefficient
        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
    for i in range(num_epoch):
        # shuffle
        idx = np.arange(num_instances)
        random.shuffle(idx)
        X = X[idx]
        y = y[idx]
        loss = 0
        for j in range(num_instances):
            t = i*num_epoch+j+1
            if alpha == '1/sqrt(t)':
                alpha = c/np.sqrt(t)
            elif alpha == '1/t':
                alpha = c/t
            elif alpha == 'new':
                alpha = c/(1+c*lambda_reg*t)

            theta_hist[i][j] = theta
            diff = np.dot(X[j],theta) - y[j]
            loss = np.dot(diff,diff) + lambda_reg * np.dot(theta,theta)
            loss_hist[i][j] = loss

            grad = 2*(np.dot(diff,X[j])+lambda_reg*theta)
            theta = theta - alpha * grad
    return theta_hist, loss_hist
```

(i) Find the best parameter c

[31]:
```
for c in [0.0001,0.001,0.01]:
    theta_hist, loss_hist=stochastic_grad_descent(X_train, y_train, alpha='new',␣
 ↪c=c)
    avg_loss_hist = np.mean(loss_hist, axis=1)
    plt.plot(range(len(avg_loss_hist)),np.log(avg_loss_hist),label="c="+str(c))
plt.title('Decaying step size for SGD (c/(1+c*lambda*t))')
plt.xlabel("epoch")
plt.ylabel("average square loss + reg")
plt.legend(loc = 1)
plt.show()
```

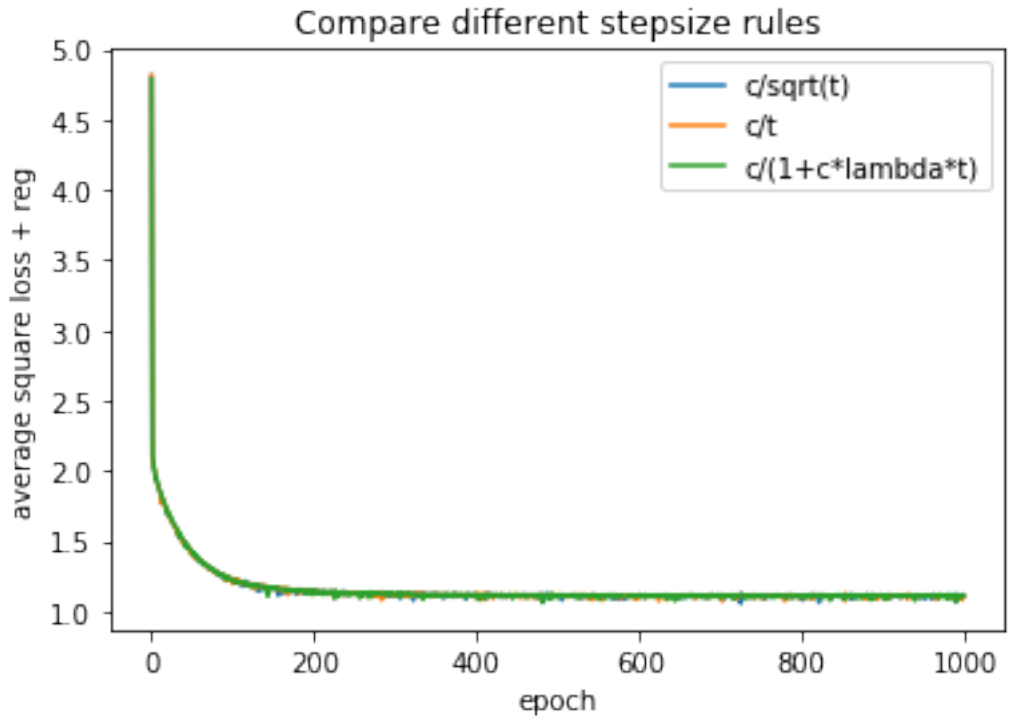


When $\eta_0 = 0.001$, the loss function converges with the minimum.

(ii) Compare different stepsize rules with their optimal parameters

[33]:
```
theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, alpha='1/
 ↪sqrt(t)', c=0.001)
avg_loss_hist = np.mean(loss_hist, axis=1)
plt.plot(range(len(avg_loss_hist)),np.log(avg_loss_hist),label='c/sqrt(t)')

theta_hist, loss_hist=stochastic_grad_descent(X_train, y_train, alpha='1/t', c=0.
 ↪001)
avg_loss_hist = np.mean(loss_hist, axis=1)
plt.plot(range(len(avg_loss_hist)),np.log(avg_loss_hist),label='c/t')
```

```
theta_hist, loss_hist=stochastic_grad_descent(X_train, y_train, alpha='new', c=0.
 ↪001)
avg_loss_hist = np.mean(loss_hist, axis=1)
plt.plot(range(len(avg_loss_hist)),np.log(avg_loss_hist),label='c/
 ↪(1+c*lambda*t)')

plt.title('Compare different stepsize rules')
plt.xlabel("epoch")
plt.ylabel("average square loss + reg")
plt.legend(loc = 1)
plt.show()
```



The three stepsize rules have similar performances.