1003 hw2

Xinmeng Li xl1575

February 2020

1 Computing Risk

1.

(a) $E[||\vec{x}||_2^2] = \sum_{i=1}^n E[x_i^2] = \sum_{i=1}^n (\frac{1}{5} * ((-2)^2 + (-1)^2 + 0^2 + 1^2 + 2^2)) = 2n$

(b) $E[||x||_\infty] = E[max_{\forall i}|x_i|] = 1 * P(max(|x_i|) = 1) + 2 * P(max(|x_i|) = 2) + 0 * P(max(|x_i|) = 0)$
= 1*P($\forall i, x_i \neq$ -2 and 2, $\exists i, x_i =$ -1 or 1) + 2*P($\exists i, x_i =$ -2 or 2)
= P($\forall i, x_i \neq$ -2 and 2) - P($\forall i, x_i = 0$) + 2 * (1 - P($\forall i, x_i \neq$ -2 and 2))
$= (\frac{3}{5})^n - (\frac{1}{5})^n + 2 - 2 * (\frac{3}{5})^n$
$= 2 - \frac{1+3^n}{5^n}$

(c) $\sum = \begin{bmatrix} Cov(x_1, x_1) & Cov(x_1, x_2) & ... & Cov(x_1, x_n) \\ Cov(x_2, x_1) & Cov(x_2, x_2) & ... & Cov(x_2, x_n) \\ ... & ... & ... & ... \\ Cov(x_n, x_1) & Cov(x_n, x_2) & ... & Cov(x_n, x_n) \end{bmatrix}$

Since x_i are independent with each other, $Cov(x_i, x_j) = 0$ for $i \neq j$. Since x_i are identically distributed, $\forall i, Cov(x_i, x_i) = Var(x_i) = E[x_i^2] - 0 = 2$.

Thus, the covariance matrix is $\sum = \begin{bmatrix} 2 & 0 & ... & 0 \\ 0 & 2 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 2 \end{bmatrix} = 2\mathbb{1}_n$

2.

(a)

Two Approaches:

Method 1: $E[(a-y)^2] = E[a^2 + y^2 - 2ay] = E[a^2] + E[y^2] - 2E[ay]$

To get a^*, we try to find the value of a when the derivative of $E[(a-y)^2]$ with respect to a is 0.

$\frac{\partial E[(a-y)^2]}{\partial a} = \frac{\partial E[a^2]}{\partial a} - \frac{2\partial E[ay]}{\partial a} = 2\int a f_Y(y)dy - 2\int y f_Y(y)dy = 0$

$\int a f_Y(y)dy = \int y f_Y(y)dy, E[a] = E[y]$. Since $E[E[y]] = E[y]$, we guess that $a^* = E[y]$.

To ensure that $E[y]$ is the MMSE estimator of y, we calculate the second derivative of $E[(a-y)^2]$ with respect to a. $\frac{\partial^2 E[(a-y)^2]}{\partial a^2} = 2\int f_Y(y)dy = 2$.

Therefore, $a^* = E[y]$.

$E[(a^* - y)^2] = E[(y - E[y])^2] = E[y^2] + E[(E[y])^2] - 2E[yE[y]] = E[y^2] + (E[y])^2 - 2(E[y])^2 = E[y^2] - (E[y])^2 = Var(y)$
Method 2: $E[(a-y)^2] = E[a^2 + y^2 - 2ay] = E[a^2] - 2E[ay] + (E[y])^2 + E[y^2] - (E[y])^2$
$= \int a^2 f_Y(y)dy - 2\int ayf_Y(y)dy + (E[y])^2 + Var(y)$
$= a^2 \int f_Y(y)dy - 2a\int yf_Y(y)dy + (E[y])^2 + Var(y)$
$= a^2 * 1 - 2a * 1 + (E[y])^2 + Var(y)$
$= (a - E[y])^2 + Var(y)$. Thus, when $a^* = E[y]$, $E[(a^* - y)^2] = Var(y)$, which is the minimum risk.

(b)
i. $E[(a-y)^2|x]$
$= E[(a - E[y|x] + E[y|x] - y)^2|x]$
$= E[(a - E[y|x])^2|x] + E[(y - E[y|x])^2|x] + 2E[(a - E[y|x])(E[y|x] - y)|x]$
$= (a - E[y|x])^2 + E[(y - E[y|x])^2|x] + 2(a - E[y|x])E[(E[y|x] - y)|x]$
$= (a - E[y|x])^2 + E[(y - E[y|x])^2|x] + 2(a - E[y|x])(E[y|x] - E[y|x])$
$= (a - E[y|x])^2 + E[(y - E[y|x])^2|x]$
Thus, when $f^*(x) = E[y|x], E[(f^*(x) - y)^2|x] = E[(y - E[y|x])^2|x] = Var(y|x)$, which is the minimum risk.
ii. By the law of iterated expectation,
$E[(f(x) - y)^2] = E[E[(f(x) - y)^2|x]]$
$= E[E[(f(x) - f^*(x) + f^*(x) - y)^2|x]]$
$= E[E[(f(x) - f^*(x))^2|x]] + E[E[(f^*(x) - y)^2|x]] + 2(f(x) - f^*(x))E[f^*(x) - E[y|x]]$
$= E[(f(x) - f^*(x))^2] + E[(f^*(x) - y)^2]$
$\geq E[(f^*(x) - y)^2]$

2 Linear Regression

2.

(a) $J(\theta) = \frac{1}{m}(X\theta - Y)^T(X\theta - Y)$
$= \frac{1}{m}(\theta^T X^T - Y^T)(X\theta - Y)$
$= \frac{1}{m}(\theta^T X^T X\theta - \theta^T X^T Y - Y^T X\theta + Y^T Y)$
Since $\theta^T X^T Y = (Y^T X\theta)^T$, and are 1x1 matrices, we have $\theta^T X^T Y = Y^T X\theta$. Thus, $J(\theta) = \frac{1}{m}(\theta^T X^T X\theta - 2\theta^T X^T Y + Y^T Y)$

(b) $\nabla J(\theta) = \frac{1}{m}(\nabla\theta^T X^T X\theta - 2\nabla\theta^T X^T Y + \nabla Y^T Y)$
$= \frac{1}{m}(2X^T X\theta - 2X^T Y)$
$= \frac{2}{m}(X^T X\theta - X^T Y)$

(c) $J(\theta + \eta h) - J(\theta) = \nabla J(\theta)\eta h = \frac{2\eta h}{m}(X^T X\theta - X^T Y)$

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

3 Ridge Regression

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

4. By using a large B, we get a small θ_0 associated with the bias. During the L2 regularization, by minimizing $\lambda||\theta||_2^2$, we prefer θ closer to 0. Therefore, a small θ_0 has little impact on the selection of the minimum $\lambda||\theta||_2^2$, and the regularization on the bias term then be reduced. To find an optimal B, we can grid search various large values of B, compare the performances on the training and validation set, and then choose the optimal B.

4 Stochastic Gradient Descent

1. $J(\theta) = \frac{1}{m}\sum_{i=1}^m (h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta$
$= \frac{1}{m}\sum_{i=1}^m (h_\theta(x_i) - y_i)^2 + \frac{m}{m}\lambda\theta^T\theta$
$= \frac{1}{m}\sum_{i=1}^m ((h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta)$
$= \frac{1}{m}\sum_{i=1}^m f_i(\theta)$
2. $E[\nabla f_i(\theta)] = \sum_{i=1}^m P(i)\nabla f_i(\theta) = \sum_{i=1}^m \frac{1}{m}\nabla f_i(\theta) = \frac{1}{m}\sum_{i=1}^m \nabla f_i(\theta) = \nabla J(\theta)$
3. $\theta = \theta - \eta\nabla J(\theta; x_i; y_i) = \theta - \eta\nabla f_i(\theta) = \theta - 2\eta x_i(h_i(\theta) - y_i) - 2\eta\lambda\theta = \theta - 2\eta(x_i^T\theta - y_i)x_i - 2\eta\lambda\theta$

Xinmeng Li xl1575 1003 hw2

In [1]:
```
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

In [2]:
```
### Assignment Owner: Tian Wang
def check_constant(arr):
    arr = arr-arr[0]
    if list(arr) == [0]*len(arr):
        return True
    return False

########################################
### Linear Regression 1(a)
### 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_normalized = train
    test_normalized = test
    ncol = train.shape[1]
    constant = []
    for i in range(ncol):
        if check_constant(train[:,i]) == True:
            constant.append(i)
            continue
        min_x = min(train[:,i])
        denom = max(train[:,i]) - min_x
        train_normalized[:,i] = (train[:,i]-min_x)/denom
        test_normalized[:,i] = (test[:,i]-min_x)/denom
    train_normalized = np.delete(train_normalized, constant, 1)
    test_normalized = np.delete(test_normalized, constant, 1)
    return train_normalized,test_normalized

########################################
### The square loss function
### Linear Regression 2(e)
########################################

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
    m = X.shape[0]
    l = np.dot(X,theta)-y
    loss = (1/m)*np.dot(l,l)
    return loss


########################################
### The gradient of the square loss function
### Linear Regression 2(f)
########################################

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)
    """
    m = X.shape[0]
    grad = (2/m)*(np.dot(X.T,np.dot(X,theta)-y))
    return grad


#########################################
### Linear Regression 3(a)
### 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
    for i in range(num_features):
        e_i =  np.array([0]*num_features)
        e_i[i] = 1
        approx_grad[i] = (compute_square_loss(X, y, theta + epsilon * e_i)-compute_square_loss(X, y, theta -ep
silon * e_i))/(2*epsilon)
    e = np.sqrt(np.sum((true_gradient-approx_grad)**2))
    #print("the Euclidean distance between the approximation and the true gradient:",e)
    if e>tolerance:
        return False
    return True


#########################################
### Linear Regression 3(a)
### 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
    """
    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
    for i in range(num_features):
        e_i = np.array([0]*num_features)
        e_i[i] = 1
        approx_grad[i] = (objective_func(X, y, theta + epsilon * e_i)-objective_func(X, y, theta -epsilon * e_
i))/(2*epsilon)
    if np.sqrt(np.sum((true_gradient-approx_grad)**2))>tolerance:
        return False
    return True



########################################
### Linear Regression 4(a)
### 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
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta)
    for i in range(1,num_step+1):
        grad = compute_square_loss_gradient(X, y, theta)
        theta = theta-alpha*grad
        theta_hist[i,:] = theta
        loss_hist[i] = compute_square_loss(X, y, theta)
        #print("theta",theta)
        if grad_check == True:
            che = grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4)
            assert(che == True),"ERROR: gradient calculator is wrong!"
    return theta_hist,loss_hist

########################################
### Linear Regression 4(c)
### Backtracking Line Search
#Check http://en.wikipedia.org/wiki/Backtracking_line_search for details
########################################
def backtrack(X,y,a,lambda_reg,r=False,num_step=1000,t=0.5):
    num_instances, num_features = X.shape[0], X.shape[1]
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    theta = np.zeros(num_features) #Initialize theta
    loss_hist[0] = compute_square_loss(X, y, theta)
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
    theta_hist[0] = theta
    alpha = a
    for i in range(1,num_step+1):
        if r == False:
            grad = compute_square_loss_gradient(X, y, theta)
        else:
            grad = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)
        loss_curr = compute_square_loss(X, y, theta-alpha*grad)
        if loss_hist[i-1] - loss_curr <= 0.5*alpha*np.dot(grad,grad):
            alpha = alpha*t
            loss_hist[i] = loss_hist[i-1]
        else:
            theta = theta-alpha*grad
```

```
            loss_hist[i] = loss_curr
            alpha = alpha/t
        theta_hist[i,:] = theta
    return theta_hist,loss_hist,alpha

#####################################
### Ridge Regression 2.
### 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)
    """
    m = X.shape[0]
    grad = (2/m)*(np.dot(X.T,np.dot(X,theta)-y))+2*lambda_reg*theta
    return grad


#####################################
### Ridge Regression 3.
### Regularized batch gradient descent
#####################################
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2, num_step=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
        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 thet
a_hist[-1]
        loss hist - the history of average square loss function without the regularization term, 1D numpy arra
y.
    """
    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
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta)
    for i in range(1,num_step+1):
        grad = compute_regularized_square_loss_gradient(X, y, theta,lambda_reg)
        theta = theta-alpha*grad
        theta_hist[i,:] = theta
        loss_hist[i] = compute_square_loss(X, y, theta)
    return theta_hist,loss_hist


#####################################
### Stochastic gradient descent 4.
#####################################
from sklearn.utils import shuffle
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_fe
atures)
                    for instance, theta in epoch 0 should be theta_hist[0], theta in epoch (num_epoch) is the
ta_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
    alpha_curr = alpha
    t = 0
    for i in range(num_epoch):
        X, y = shuffle(X, y)
        for j in range(num_instances):
            t = t+1
            if alpha == "1/sqrt(t)":
                alpha_curr = C/np.sqrt(t)
            elif alpha == "1/t":
                alpha_curr = C/t
            grad = 2*X[j]*(np.dot(X[j],theta)-y[j])+2*lambda_reg*theta
            theta = theta - alpha_curr*grad
            theta_hist[i][j] = theta
            loss = compute_square_loss(X, y, theta)+lambda_reg*np.dot(theta,theta)
            loss_hist[i][j] = loss
    return theta_hist, loss_hist

########################################
### Stochastic gradient descent 6(b).
### Try a new step size rule.
########################################
def stochastic_grad_descent_6b(X, y, alpha=0.01, lambda_reg=10**-2, num_epoch=1000,C=0.1):
    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
    t = 0
    alpha_curr = alpha
    for i in range(num_epoch):
        X, y = shuffle(X, y)
        for j in range(num_instances):
            t = t+1
            alpha_curr = alpha/(1+alpha*lambda_reg*t)
            grad = 2*X[j]*(np.dot(X[j],theta)-y[j])+2*lambda_reg*theta
            theta = theta - alpha_curr*grad
            theta_hist[i][j] = theta
            loss = compute_square_loss(X, y, theta)+lambda_reg*np.dot(theta,theta)
            loss_hist[i][j] = loss
    return theta_hist, loss_hist
```

In [3]:

```
########################################
### Verify the correctness
### Linear Regression 2(e)
########################################
A = np.random.rand(3,5)
b = np.random.rand(3)
t = np.random.rand(5)
print("Our compute_square_loss function has result:")
print(compute_square_loss(A, b, t))
from sklearn.metrics import mean_squared_error
print("The sklearn mse function has result:")
print(mean_squared_error(b,np.dot(A,t)))
```

```
Our compute_square_loss function has result:
1.1373115756133159
The sklearn mse function has result:
1.1373115756133159
```

In [4]:
```
########################################
### Verify the correctness
### Linear Regression 2(f)
########################################
A = np.random.rand(1,2)
b = np.random.rand(1,1)
t = np.random.rand(2,1)
print("Our compute_square_loss_gradient function has result:")
print(compute_square_loss_gradient(A, b, t))
print("Result of another way to compute the gradient by hand:")
print((np.dot(np.dot(A.T,A),t)-A.T*b[0,0])*2)
```

```
Our compute_square_loss_gradient function has result:
[[-0.85891201]
 [-0.17039815]]
Result of another way to compute the gradient by hand:
[[-0.85891201]
 [-0.17039815]]
```

In [5]:
```
#Loading the dataset
print('loading the dataset')
df = pd.read_csv('/home/jovyan/shared/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)
```

```
loading the dataset
Split into Train and Test
```

In [6]:
```
X_train.shape
```

Out[6]:
```
(100, 48)
```

In [7]:
```
print("Scaling all to [0, 1]")
X_train, X_test= feature_normalization(X_train, X_test)
```

```
Scaling all to [0, 1]
```

In [8]:
```
X_train = np.hstack((X_train, np.ones((X_train.shape[0], 1))))  # Add bias term
X_test = np.hstack((X_test, np.ones((X_test.shape[0], 1))))  # Add bias term
```

In [9]:
```
########################################
### Linear Regression 3(a)
### Gradient Checker
########################################
theta_hist,loss_hist = batch_grad_descent(X_train, y_train, alpha=0.01, grad_check=True)
```

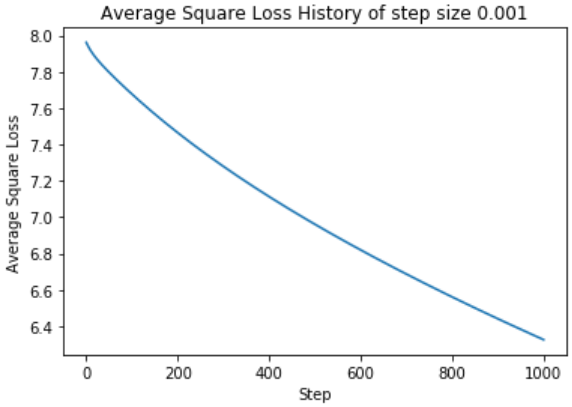
In [10]:
```
theta_hist,loss_hist = batch_grad_descent(X_train, y_train, alpha=0.5, grad_check=True)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-10-eb36b5719fef> in <module>
----> 1 theta_hist,loss_hist = batch_grad_descent(X_train, y_train, alpha=0.5, grad_check=True)

<ipython-input-2-abb86d4f4219> in batch_grad_descent(X, y, alpha, num_step, grad_check)
    197         if grad_check == True:
    198             che = grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4)
--> 199             assert(che == True),"ERROR: gradient calculator is wrong!"
    200     return theta_hist,loss_hist
    201

AssertionError: ERROR: gradient calculator is wrong!
```

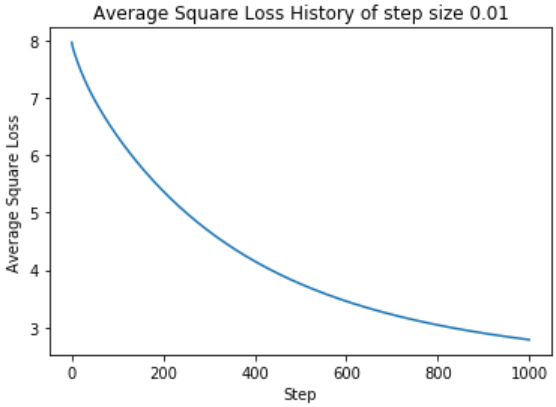
We observe that when alpha is 0.5, the gradient exploded and thus the error is raised. When the value of alpha is appropriate such as 0.01, no error thrown, thus the gradient was calculated correctly.

In [11]:

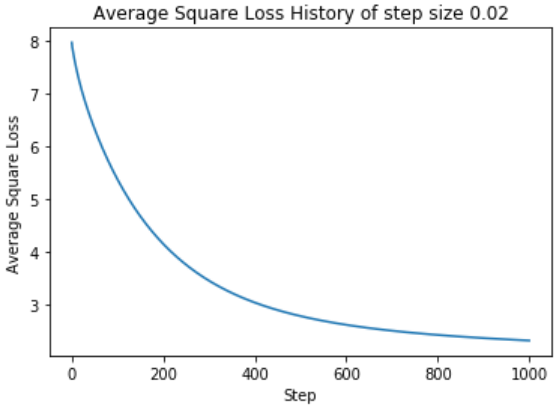
```
########################################
### Linear Regression 4(b)
### Batch gradient descent
########################################
import time
bgd_time = []
avg_loss = []
steps = [0.001,0.01,0.02,0.03,0.04,0.05,0.06,0.1,0.5]
for s in steps:
    start = time.time()
    theta_hist,loss_hist = batch_grad_descent(X_train, y_train, alpha=s, grad_check=False)
    bgd_time.append(time.time()-start)
    avg_loss.append(loss_hist)

for i in range(len(steps)):
    plt.plot(np.arange(1001),avg_loss[i])
    plt.ylabel("Average Square Loss")
    plt.xlabel("Step")
    plt.title("Average Square Loss History of step size "+str(steps[i]))
    plt.show()
    print("The final loss is ",avg_loss[i][-1])
```



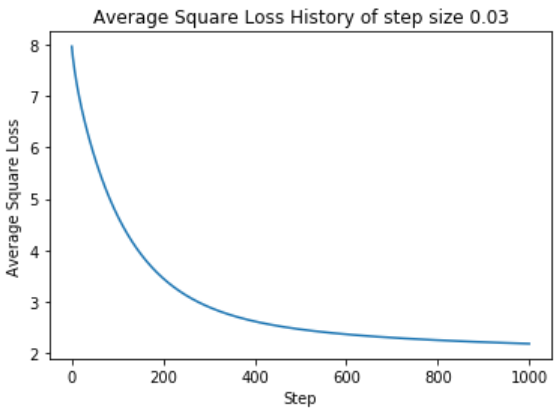

```
The final loss is  6.325376024793016
```



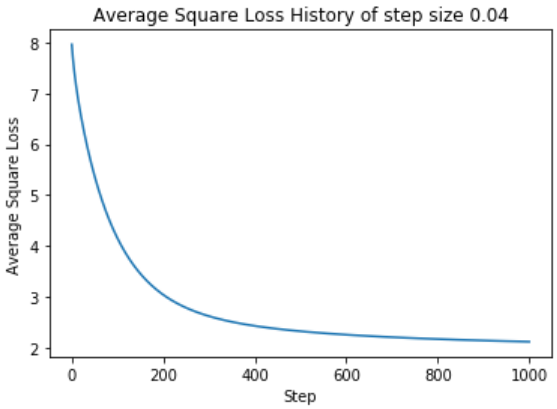

```
The final loss is  2.7854584187675564
```



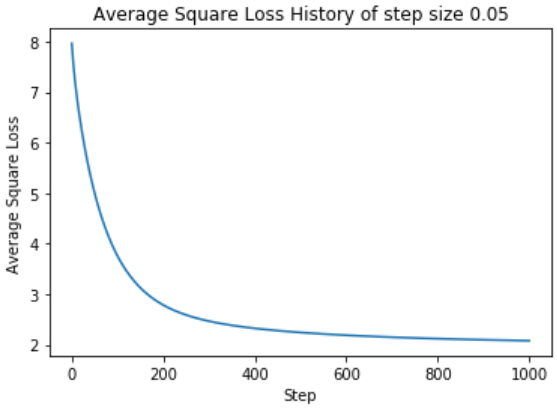

```
The final loss is  2.3225696673704923
```



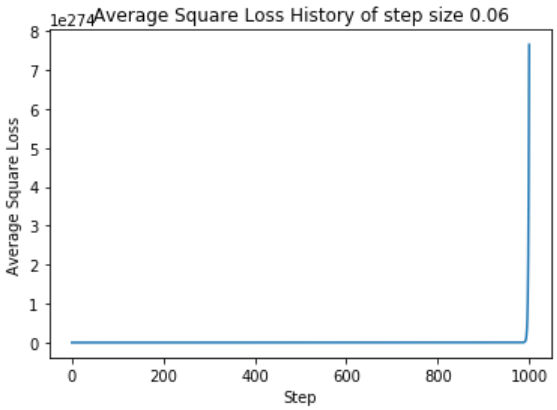

```
The final loss is  2.1859162241629586
```



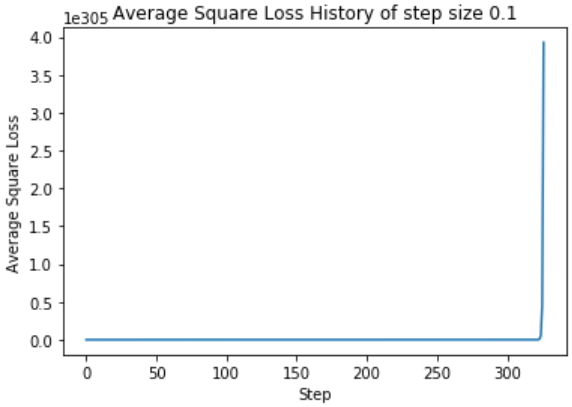

```
The final loss is  2.1172917841648524
```



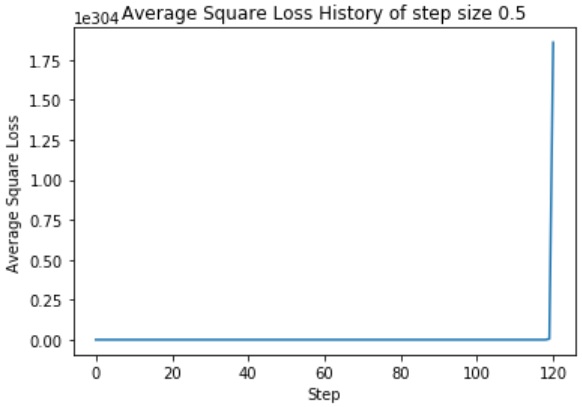

```
The final loss is  2.0776993701242232
```




```
The final loss is  7.646913298674167e+274
```




```
The final loss is  nan
```

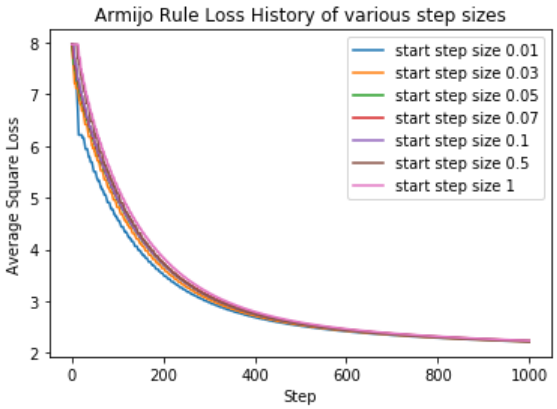



```
The final loss is  nan
```

We observe that when step size > 0.05, the average square loss exploded. When step size <=0.05, as the step size increases, the loss converges faster, the final step loss decreases. 0.05 is the optimal fixed step size due to its lowest loss.

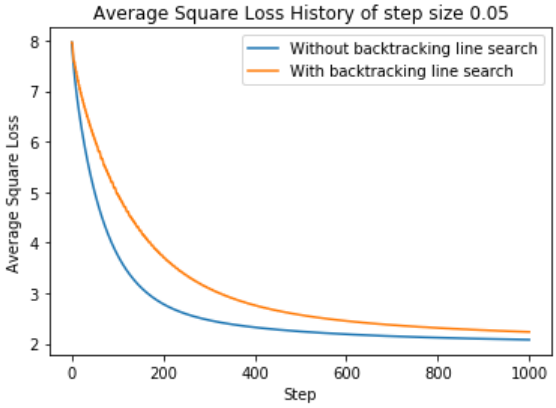
In [12]:
```
########################################
### Linear Regression 4(c)
### Backtracking Line Search
########################################
step_list = [0.01,0.03,0.05,0.07,0.1,0.5,1]
back_loss = []
bls_time = []
for s in step_list:
    start = time.time()
    theta,loss_hist,alpha = backtrack(X_train,y_train,a=s,lambda_reg = 0,t=0.8)
    bls_time.append(time.time()-start)
    back_loss.append(loss_hist)
    plt.ylabel("Average Square Loss")
    plt.xlabel("Step")
    plt.title("Armijo Rule Loss History of various step sizes")
    plt.plot(loss_hist,label="start step size "+str(s))
    print("The final step size for the start step size", s,"is",alpha,",The final loss is ",loss_hist[-1])
plt.legend()
plt.show()
```

```
The final step size for the start step size 0.01 is 0.059604644775390625 ,The final loss is  2.21865611301670
37
The final step size for the start step size 0.03 is 0.046874999999999986 ,The final loss is  2.22051320750323
94
The final step size for the start step size 0.05 is 0.05000000000000002 ,The final loss is  2.23148316363957
The final step size for the start step size 0.07 is 0.07 ,The final loss is  2.2340664786902176
The final step size for the start step size 0.1 is 0.06400000000000002 ,The final loss is  2.22900322962622
The final step size for the start step size 0.5 is 0.05368709120000003 ,The final loss is  2.2259468029338167
The final step size for the start step size 1 is 0.06871947673600005 ,The final loss is  2.2381724656317705
```



We observe that with backtracking line search, the loss explosion is eliminated. The average square loss converges even when the start step size is 1.

In [32]:
```
plt.plot(avg_loss[5],label = "Without backtracking line search")
plt.plot(back_loss[2],label = "With backtracking line search")
plt.ylabel("Average Square Loss")
plt.xlabel("Step")
plt.legend()
plt.title("Average Square Loss History of step size "+str(steps[5]))
plt.show()
```



Comparing with the best fixed step size 0.05, gradient descent with backtracking line search converges slower. In each iteration, the backtracking has one more if statement and one more step size update, which will take extra time. However, the time for gradient descent with or without backtracking line search is O(n), where n is the number of step. Therefore, theoritically the runtime for these two algorithms are close and backtracking line search is a little bit slower. Now let's take a look at the time they have taken in practice.

In [14]:
```
print(bgd_time)
```

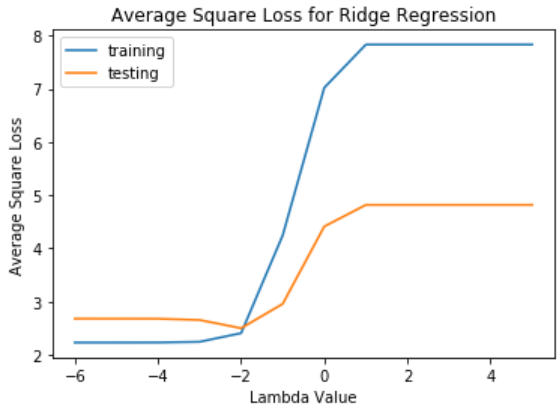
```
[0.018984556198120117, 0.01648879051208496, 0.016724586486816406, 0.0157930850982666, 0.015876293182373047,
0.015924692153930664, 0.025891780853271484, 0.021693944931030273, 0.015827417373657227]
```

In [15]:
```
print(bls_time)
```

```
[0.023802518844604492, 0.022860050201416016, 0.02079463005065918, 0.02198934555053711, 0.021509408950805664,
0.025249481201171875, 0.022571086883544922]
```

Obviously, backtracking line search is a bit slower and the time for these two algorithms have the same order of magnitude, i.e. 10^{-2}, which verifies our guess.

In [16]:
```
#######################################
### Ridge Regression 5.
### Find the optimal lambda value.
#######################################
theta_train = []
train_loss = []
test_loss = []
lam = [10**i for i in range(-6,6)]
for l in lam:
    # Here we set the r parameter to be True, which means the backtrack algorithm
    # will use ridge regression for gradient update.
    theta,loss,alpha = backtrack(X_train,y_train,a=0.05,t=0.8,r = True,lambda_reg=l)
    theta_train.append(theta)
    train_loss.append(compute_square_loss(X_train, y_train, theta[-1]))
    test_loss.append(compute_square_loss(X_test, y_test, theta[-1]))

plt.title("Average Square Loss for Ridge Regression")
plt.ylabel("Average Square Loss")
plt.xlabel("Lambda Value")
plt.plot(np.arange(-6,6),train_loss,label = "training")
plt.plot(np.arange(-6,6),test_loss,label = "testing")
plt.legend()
plt.show()
```



We observe that when λ is approximately 0.01, the curve of training and testing loss intersects, which indicates that the model has consistent performance on both the training and testing dataset. Moreover, the testing loss also is the lowest when λ = 0.01. Therefore, we choose λ = 0.01 as a candidate for the optimal λ value. Now let's take a look at its loss history.

In [17]:
```
print(train_loss)
```

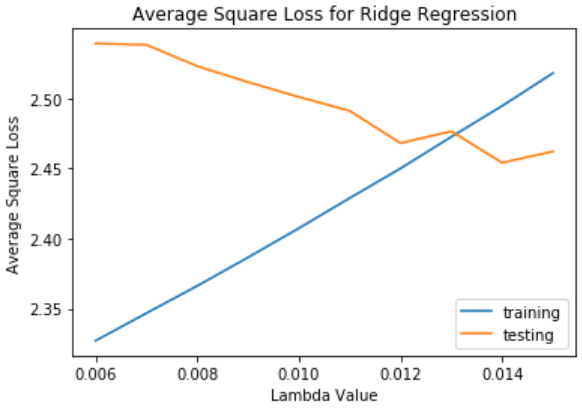
```
[2.2314960458006836, 2.23161204058704, 2.2326131506033384, 2.244803241121335, 2.407377237664881, 4.246149050350842, 7.020825754982518, 7.831422679928133, 7.831422679928236, 7.831422679928236, 7.831422679928236, 7.831422679928236]
```

In [18]:
```
print(test_loss)
```

```
[2.679941509304512, 2.679696259189303, 2.6785018007006363, 2.6551341636904033, 2.501012026407914, 2.9578828082299826, 4.413239170399601, 4.818541553888523, 4.818541553888349, 4.818541553888349, 4.818541553888349, 4.818541553888349]
```

After printing the training and testing loss, we observe that the model do have the best performance on the test set when λ = 0.01. Also the training and testing loss are very close when λ = 0.01. Now let's try more λ value around 0.01.

In [19]:
```
theta_train = []
train_loss = []
test_loss = []
lam = np.arange(0.006,0.016,0.001)
for l in lam:
    theta,loss,alpha = backtrack(X_train,y_train,a=0.05,t=0.8,r = True,lambda_reg=l)
    theta_train.append(theta[-1])
    train_loss.append(compute_square_loss(X_train, y_train, theta[-1]))
    test_loss.append(compute_square_loss(X_test, y_test, theta[-1]))

plt.title("Average Square Loss for Ridge Regression")
plt.ylabel("Average Square Loss")
plt.xlabel("Lambda Value")
plt.plot(lam,train_loss,label = "training")
plt.plot(lam,test_loss,label = "testing")
plt.legend()
plt.show()
```



In [20]:
```
print(train_loss)
```

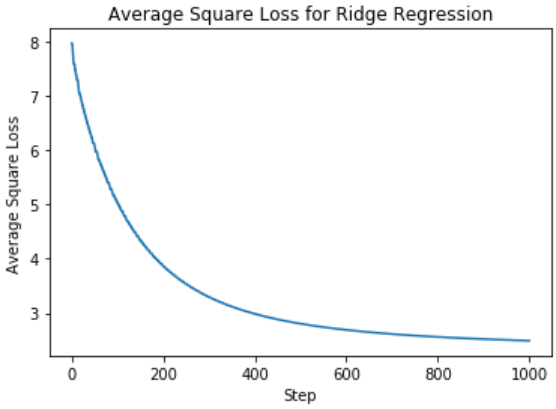
```
[2.32714488767918, 2.346790176733897, 2.3662603898242445, 2.386574139325548, 2.407377237664881, 2.42886215625
98274, 2.4500358515416654, 2.4725550529232105, 2.4946191259980273, 2.5180024568193535]
```

In [21]:
```
print(test_loss)
```

```
[2.5391355816130474, 2.5382034099999564, 2.5229454396075224, 2.5116031500399933, 2.501012026407914, 2.4910829
068683387, 2.4680884891798454, 2.4765147123594295, 2.4541018921853643, 2.4621341098521152]
```

Since when λ = 0.014, the training loss 2.495 is low and the testing loss 2.454 is the lowest, we choose this as our optimal λ value.

In [22]:
```
########################################
### Ridge Regression 6.
### Select a theta.
########################################
theta,loss,alpha = backtrack(X_train,y_train,a=0.05,t=0.8,r = True,lambda_reg=0.014)
plt.title("Average Square Loss for Ridge Regression")
plt.ylabel("Average Square Loss")
plt.xlabel("Step")
plt.plot(loss)
plt.show()
```



In [23]:
```
print(loss[-5:])
```

```
[2.49610483 2.49558693 2.49558693 2.49508455 2.49461913]
```

We observe that as iteration increases, the loss decreases, which means the ASL curve converges. Thus, the last θ is the one which minimizes the loss.

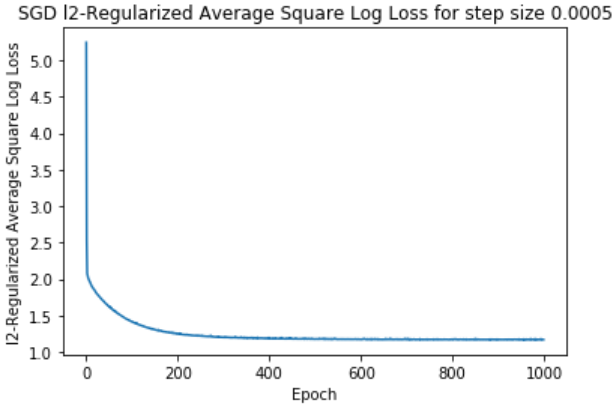
In [24]:
```
print(theta[-1])
```

```
[-1.22261299  0.53906274  1.43640989  2.40933131 -1.83083635 -0.79800977
 -0.81985964 -0.81985964  0.76742013  1.42178423  2.48107298 -0.4653166
 -1.34059867 -3.97719494  1.50225224  2.42272729  1.29601068  0.36618343
 -0.08038093 -0.08038093 -0.08038093 -0.01687786 -0.01687786 -0.01687786
  0.00994494  0.00994494  0.00994494  0.02311738  0.02311738  0.02311738
  0.03066188  0.03066188  0.03066188 -0.06123512 -0.06123512 -0.06123512
  0.08012543  0.08012543  0.08012543  0.06713617  0.06713617  0.06713617
  0.06126947  0.06126947  0.06126947  0.05805648  0.05805648  0.05805648
 -1.24412758]
```

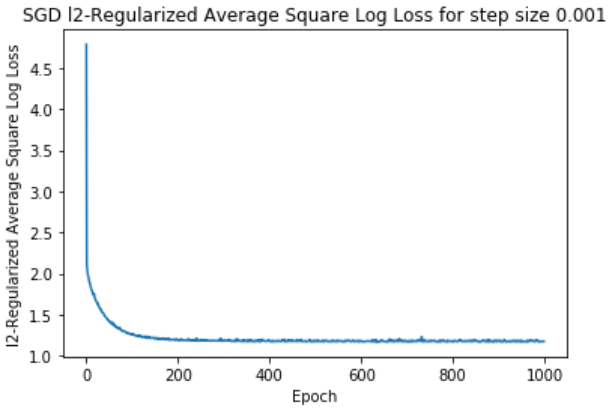
I would choose λ = 0.014 and the above θ in the practice.

In [37]:
```
#######################################
### Stochastic gradient descent 5.
### Try various fixed step sizes.
#######################################
for a in [0.0005,0.001,0.005,0.01,0.05]:
    theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, lambda_reg = 0.014,alpha=a)
    plt.title("SGD l2-Regularized Average Square Log Loss for step size "+str(a))
    plt.ylabel("l2-Regularized Average Square Log Loss")
    plt.xlabel("Epoch")
    plt.plot(np.log([np.mean(l) for l in loss_hist]))
    print("the minimum of the mean loss of a whole epoch is",min([np.mean(l) for l in loss_hist]))
    plt.show()
```

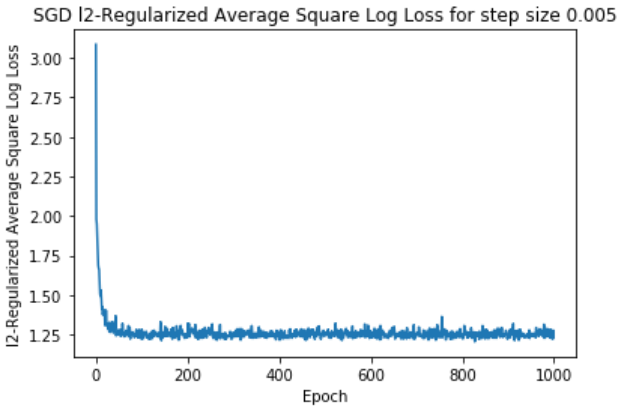
the minimum of the mean loss of a whole epoch is 3.2195492624968653



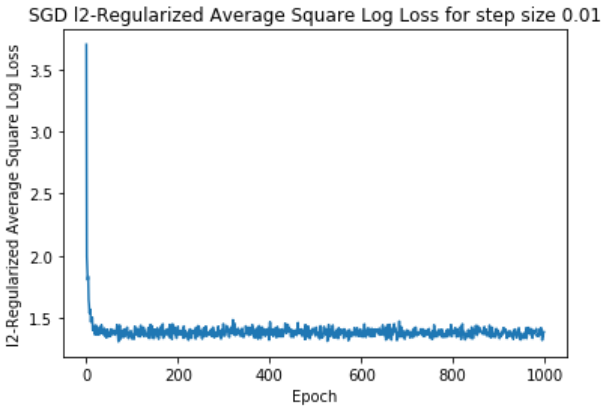
the minimum of the mean loss of a whole epoch is 3.2219891995787164



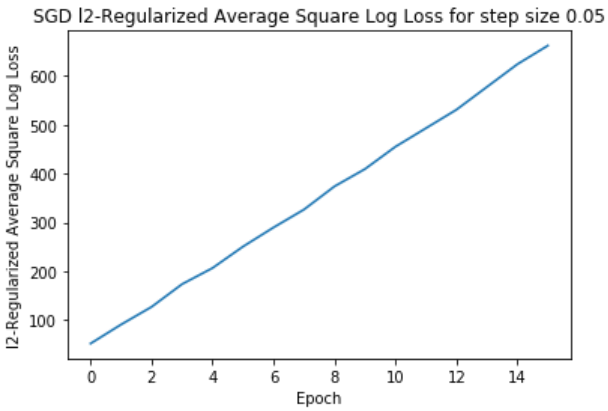
the minimum of the mean loss of a whole epoch is 3.3349930021162697



the minimum of the mean loss of a whole epoch is 3.7003695316070258



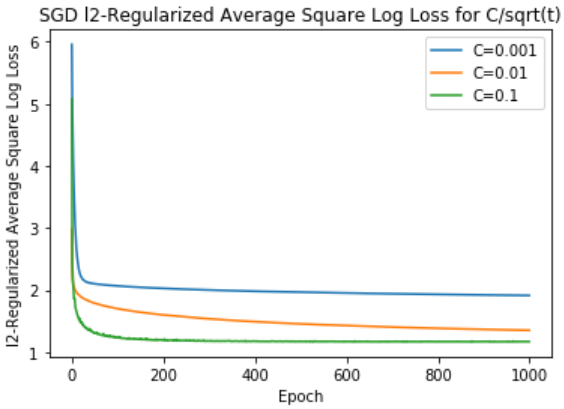
```
/opt/conda/envs/dsga-1003/lib/python3.7/site-packages/ipykernel_launcher.py:323: RuntimeWarning: overflow enc
ountered in multiply

the minimum of the mean loss of a whole epoch is 2.3118417464288904e+22
```



We observe that when the fixed step size is 0.05, the loss explode. When the step size < 0.05, as the step size increases, the fluctuations of the regularized ASL becomes stronger, the minimum of the epoch average regularized ASL increases. Thus, the optimal fixed step size is 0.0005

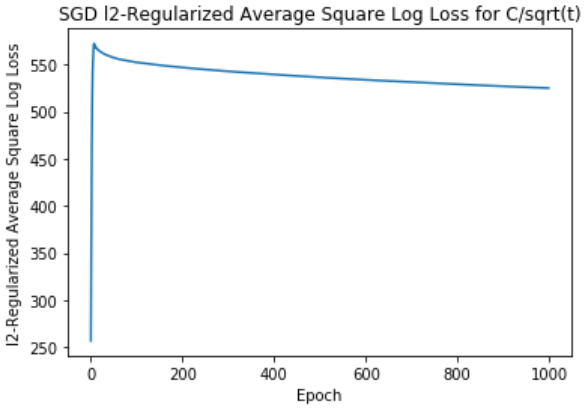
In [40]:
```
########################################
### Stochastic gradient descent 5.
### Try various C values for 1/sqrt(t).
########################################
c_val = [10**i for i in range(-3,0)]
plt.title("SGD l2-Regularized Average Square Log Loss for C/sqrt(t)")
plt.ylabel("l2-Regularized Average Square Log Loss")
plt.xlabel("Epoch")
for c in c_val:
    theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, lambda_reg = 0.014,alpha="1/sqrt(t)",C=c
)
    plt.plot(np.log([np.mean(l) for l in loss_hist]),label = "C="+str(c))
    print("When alpha=C/sqrt(t), C=",c,"the minimum of the mean loss of a whole epoch is",min([np.mean(l) for
l in loss_hist]))
plt.legend()
plt.show()
```

```
When alpha=C/sqrt(t), C= 0.001 the minimum of the mean loss of a whole epoch is 6.772052497184842
When alpha=C/sqrt(t), C= 0.01 the minimum of the mean loss of a whole epoch is 3.8639705334828367
When alpha=C/sqrt(t), C= 0.1 the minimum of the mean loss of a whole epoch is 3.215623277311467
```



In [41]:
```
plt.title("SGD l2-Regularized Average Square Log Loss for C/sqrt(t)")
plt.ylabel("l2-Regularized Average Square Log Loss")
plt.xlabel("Epoch")
theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, lambda_reg = 0.014,alpha="1/sqrt(t)",C=1)
plt.plot(np.log([np.mean(l) for l in loss_hist]),label = "C="+str(1))
print("When alpha=C/sqrt(t), C=",1,"the minimum of the mean loss of a whole epoch is",min([np.mean(l) for l in
loss_hist]))
plt.show()
```

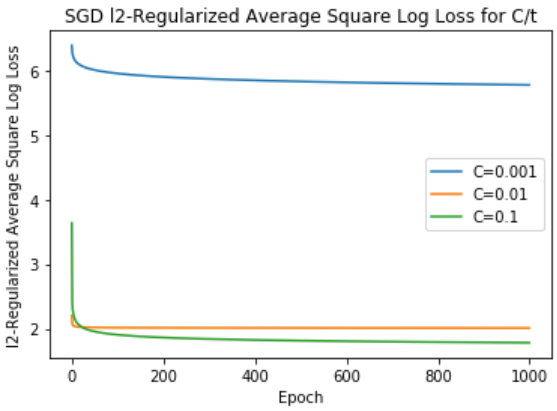
```
When alpha=C/sqrt(t), C= 1 the minimum of the mean loss of a whole epoch is 4.132182237311056e+111
```



We observe that when C=1, the regularized loss explode. When C < 1, as C increases, although the loss converges slower, the minimum of the epoch average regularized ASL decreases. Thus, the optimal C value for C/sqrt(t) is 0.1.

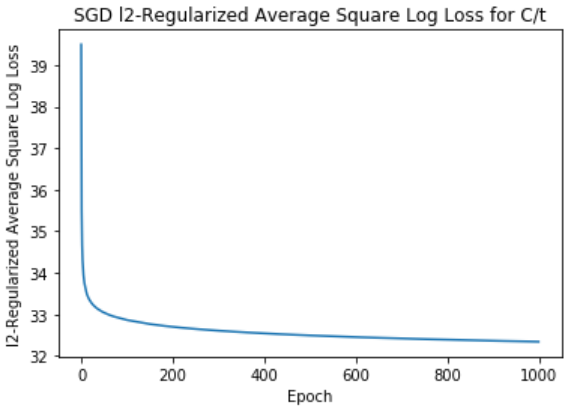
In [42]:
```
########################################
### Stochastic gradient descent 5.
### Try various C values for 1/t.
########################################
plt.title("SGD l2-Regularized Average Square Log Loss for C/t")
plt.ylabel("l2-Regularized Average Square Log Loss")
plt.xlabel("Epoch")
for c in c_val:
    theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, lambda_reg = 0.014,alpha="1/t",C=c)
    plt.plot(np.log([np.mean(l) for l in loss_hist]),label = "C="+str(c))
    print("When alpha=C/t, C=",c,"the minimum of the mean loss of a whole epoch is",min([np.mean(l) for l in l
oss_hist]))
plt.legend()
plt.show()
```

```
When alpha=C/t, C= 0.001 the minimum of the mean loss of a whole epoch is 323.8430271429036
When alpha=C/t, C= 0.01 the minimum of the mean loss of a whole epoch is 7.453546057313148
When alpha=C/t, C= 0.1 the minimum of the mean loss of a whole epoch is 5.935499871547106
```



In [43]:
```
plt.title("SGD l2-Regularized Average Square Log Loss for C/t")
plt.ylabel("l2-Regularized Average Square Log Loss")
plt.xlabel("Epoch")
theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, lambda_reg = 0.014,alpha="1/t",C=1)
plt.plot(np.log([np.mean(l) for l in loss_hist]),label = "C="+str(1))
print("When alpha=C/t, C=",1,"the minimum of the mean loss of a whole epoch is",min([np.mean(l) for l in loss_
hist]))
plt.show()
```

```
When alpha=C/t, C= 1 the minimum of the mean loss of a whole epoch is 111032422086982.36
```



We observe that when C=1, the regularized loss explode. When C < 1, as C increases, the minimum of the epoch average regularized ASL decreases. Thus, the optimal C value for C/t is 0.1.

In conclusion, the regularized ASLs of alpha = C/t are generally higher than those of alpha = C/sqrt(t). Although with a small fixed step size a low loss can be achieved, the regularized ASL curve have some fluntuations in this case, which indicates a slower convergence. Thus, 0.1/sqrt(t) is the optimal step size rule.

In [28]:
```
########################################
### Stochastic gradient descent 6(a).
### Compare the average theta with the previous theta.
########################################
def avg_theta(theta_hist):
    theta = np.zeros(theta_hist.shape[2])
    for i in range(theta_hist.shape[0]):
        for j in range(theta_hist.shape[1]):
            theta = theta+theta_hist[i][j]
    theta = theta/(theta_hist.shape[0]*theta_hist.shape[1])
    return theta

theta_hist1, loss_hist1 = stochastic_grad_descent(X_train, y_train, lambda_reg = 0.014,alpha="1/sqrt(t)",C=0.1
)
theta_hist2, loss_hist2 = stochastic_grad_descent(X_test, y_test, lambda_reg = 0.014,alpha="1/sqrt(t)",C=0.1)
theta1 = theta_hist1[-1][-1]
theta2 = theta_hist2[-1][-1]
theta3 = avg_theta(theta_hist1)
theta4 = avg_theta(theta_hist2)
train_loss1 = compute_square_loss(X_train, y_train, theta1)
test_loss1 = compute_square_loss(X_test, y_test, theta2)
train_loss2 = compute_square_loss(X_train, y_train, theta3)
test_loss2 = compute_square_loss(X_test, y_test, theta4)
print("Training Loss for theta_t",train_loss1)
print("Testing Loss for theta_t",test_loss1)
print("Training Loss for the average theta",train_loss2)
print("Testing Loss for the average theta",test_loss2)
```

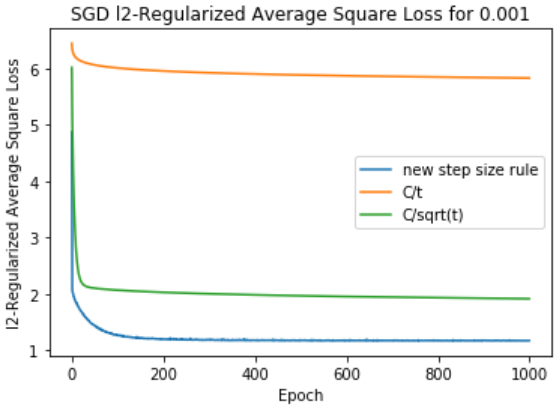
```
Training Loss for theta_t 2.3867311839856407
Testing Loss for theta_t 2.165839274185514
Training Loss for the average theta 2.526824017629833
Testing Loss for the average theta 2.2230053951438102
```

We observe that the average theta performs slightly worse than the last theta on both the training set and the test set. Normally, we assume the average theta has a better performance on the test set, because we usually use the average theta to avoid overfitting and to better generalize on the test set. However, in this case, SGD has a high variance, which makes it harder for the average theta to catch the essential information.

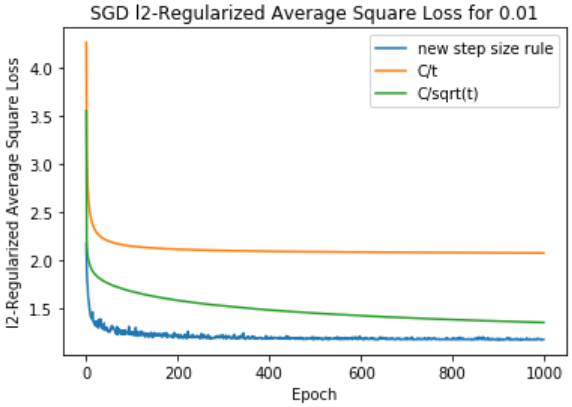
In [48]:

```
#######################################
### Stochastic gradient descent 6(b).
### Try a new step size rule.
#######################################
for c in c_val:
    theta_hist, loss_hist = stochastic_grad_descent_6b(X_train, y_train, lambda_reg = 0.014,alpha=c)
    plt.plot(np.log([np.mean(l) for l in loss_hist]),label = "new step size rule")
    print("New step size rule: the minimum of the mean loss of a whole epoch is",min([np.mean(l) for l in loss
_hist]))
    theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, lambda_reg = 0.014,alpha="1/t",C=c)
    plt.plot(np.log([np.mean(l) for l in loss_hist]),label = "C/t")
    print("C/t: the minimum of the mean loss of a whole epoch is",min([np.mean(l) for l in loss_hist]))
    theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, lambda_reg = 0.014,alpha="1/sqrt(t)",C=c
)
    plt.plot(np.log([np.mean(l) for l in loss_hist]),label = "C/sqrt(t)")
    print("C/sqrt(t): the minimum of the mean loss of a whole epoch is",min([np.mean(l) for l in loss_hist]))
    plt.title("SGD l2-Regularized Average Square Loss for "+str(c))
    plt.ylabel("l2-Regularized Average Square Loss")
    plt.xlabel("Epoch")
    plt.legend()
    plt.show()
```

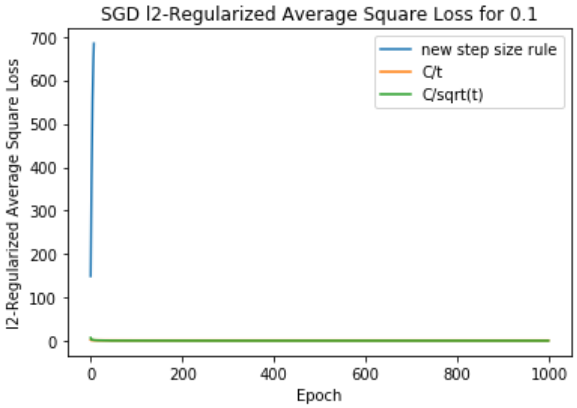
```
New step size rule: the minimum of the mean loss of a whole epoch is 3.2163917121641674
C/t: the minimum of the mean loss of a whole epoch is 341.25307649612114
C/sqrt(t): the minimum of the mean loss of a whole epoch is 6.779614241849391
```




```
New step size rule: the minimum of the mean loss of a whole epoch is 3.2186355640906608
C/t: the minimum of the mean loss of a whole epoch is 7.950285103695649
C/sqrt(t): the minimum of the mean loss of a whole epoch is 3.8520148688331495
```




```
New step size rule: the minimum of the mean loss of a whole epoch is 8.047295656883554e+64
C/t: the minimum of the mean loss of a whole epoch is 6.863220175994766
C/sqrt(t): the minimum of the mean loss of a whole epoch is 3.216642409554555
```



We observe that when η_0 has a small value < 0.1, the new step size rule has the lowest regularized ASL and converges relatively fast. When η_0 >=0.1, the regularized ASL for the new step size rule explodes.