In [5]:

```
import pandas as pd
import logging
import numpy as np
import sys
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cross_validation import train_test_split
```

```
C:\Users\liuyt\Anaconda3\lib\site-packages\sklearn\cross_validation.py:44: Deprecati
onWarning: This module was deprecated in version 0.18 in favor of the model_selectio
n module into which all the refactored classes and functions are moved. Also note th
at the interface of the new CV iterators are different from that of this module. Thi
s module will be removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

# Question2.1

In [47]:

```
### Assignment Owner: Tian Wang
#######################################
#### 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 1the training set.

    Args:
        train - training set, a 2D numpy array of size (num_instances, num_features)
        test  - test set, a 2D numpy array of size (num_instances, num_features)
    Returns:
        train_normalized - training set after normalization
        test_normalized  - test set after normalization

    """
    # TODO
    #train = np.array(train,dtype=int)
    max_train = train.max(axis=0)
    min_train = train.min(axis=0)
    train_normalized = (train-min_train)/(max_train-min_train)
    test_normalized = (test-min_train)/(max_train-min_train)
    return train_normalized, test_normalized
```

In [ ]:

In [ ]:

# Question 2.2

## 2.2.1

### The expression for J

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

## 2.2.2

### The gradient of J

$$\nabla J(\theta) = \frac{1}{m} \cdot X^T \cdot (X \cdot \theta - y)$$

## 2.2.3

$$J(\theta + \eta\Delta) - J(\theta) = \eta \cdot \nabla^T J(\theta) \cdot \Delta$$

## 2.2.4

$$\theta = \theta - \eta * \nabla J(\theta)$$

## 2.2.5

In [7]:

```
########################################
#### The square loss function
def compute_square_loss(X, y, theta):
    """
    Given a set of X, y, theta, compute the 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 square loss, scalar
    """
    loss = 0 #initialize the square_loss
    #TODO
    residual = np.dot(X,theta)-y
    residual2 = residual**2
    loss = np.sum(residual2)/(2*len(theta))
    return loss
```

In [8]:

```
X=np.array([[2,3,4],[1,2,1],[4,5,4]])
y=np.array([4,3,2])
theta=np.array([1,1,1])
compute_square_loss(X,y,theta)
```

Out[8]:

24.5

# 2.2.6

In [9]:

```
###########################################
### compute the gradient of square loss function
def compute_square_loss_gradient(X, y, theta):
    """
    Compute gradient of the 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
    grad = np.zeros(X.shape[1])
    grad = np.dot(X.T,np.dot(X,theta)-y)/X.shape[0]
    #for i in range(X.shape[1]):
    #    grad[i]=sum((np.dot(X,theta)-y)*X[:,i])/X.shape[0]
    return grad
```

In [10]:

```
compute_square_loss_gradient(X,y,theta)
```

Out[10]:

array([ 18.33333333,  24.        ,  21.66666667])

# Question 2.3

In [11]:

```
###############################################
### 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 indicate 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
    eps_matrix = np.diag(np.ones(num_features))
    print (eps_matrix)
    for i in range(num_features):
        print(eps_matrix[i])
        J_diff_i = compute_square_loss(X, y, theta+epsilon*eps_matrix[i])-compute_square_loss(X, y,
        grad_i = J_diff_i/(2*epsilon)
        approx_grad[i] = grad_i

    sum_square=np.sum((true_gradient-approx_grad)**2)
    eclid = np.sqrt(sum_square)
    if eclid<=tolerance:
        return True
    else:
        return False
```

In [12]:

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

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
[ 1.  0.  0.]
[ 0.  1.  0.]
[ 0.  0.  1.]
```

Out[12]:

```
True
```

# Question 2.4

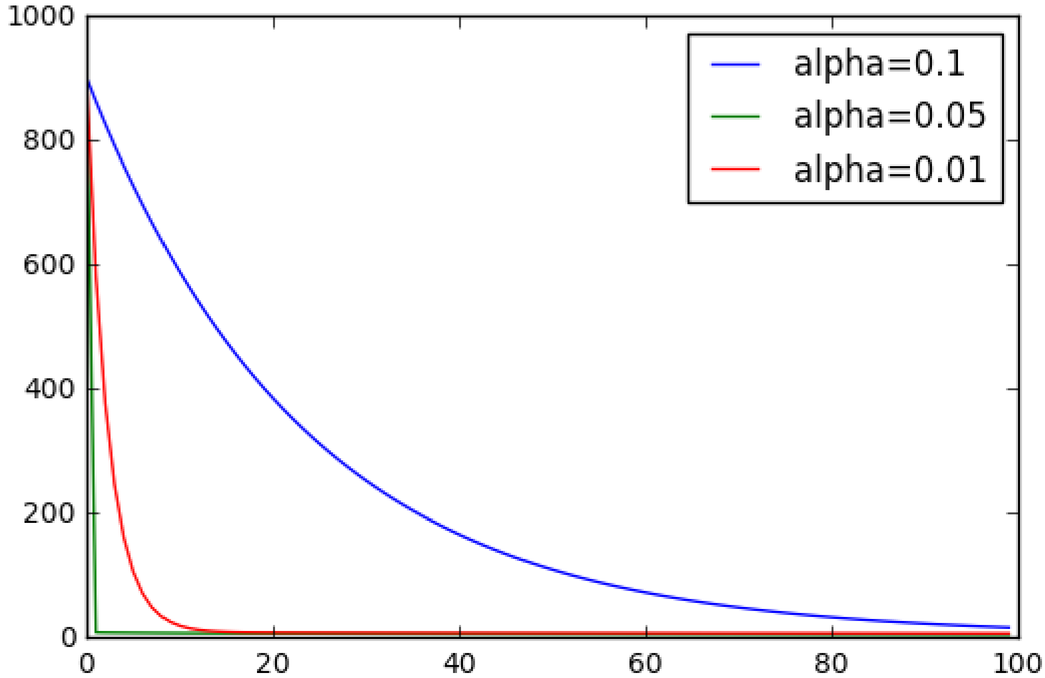
In [13]:

```
from sklearn.model_selection import train_test_split
import pandas as pd

print('loading the dataset')
df = pd.read_csv('D:\Spring_2017\machine learning\homework\hw1\hw1-sgd\hw1-data.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(X_train.shape)
print("Scaling all to [0, 1]")
X_train, X_test = feature_normalization(X_train, X_test)
print(X_train.shape)
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

####################################
#### Batch Gradient Descent
def batch_grad_descent(X, y, alpha=0.1, num_iter=1000, check_gradient=False):
    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.ones(num_features) #initialize theta
    #TODO
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta_hist[0])
    #print(theta_hist[0])
    for i in range(1,num_iter+1):
        theta_hist[i] = theta_hist[i-1]-alpha*compute_square_loss_gradient(X, y, theta_hist[i-1])
        #print(theta_hist[i])
        loss_hist[i] = compute_square_loss(X, y, theta_hist[i])
    return theta_hist, loss_hist
```

```
loading the dataset
Split into Train and Test
(100, 48)
Scaling all to [0, 1]
(100, 48)
```

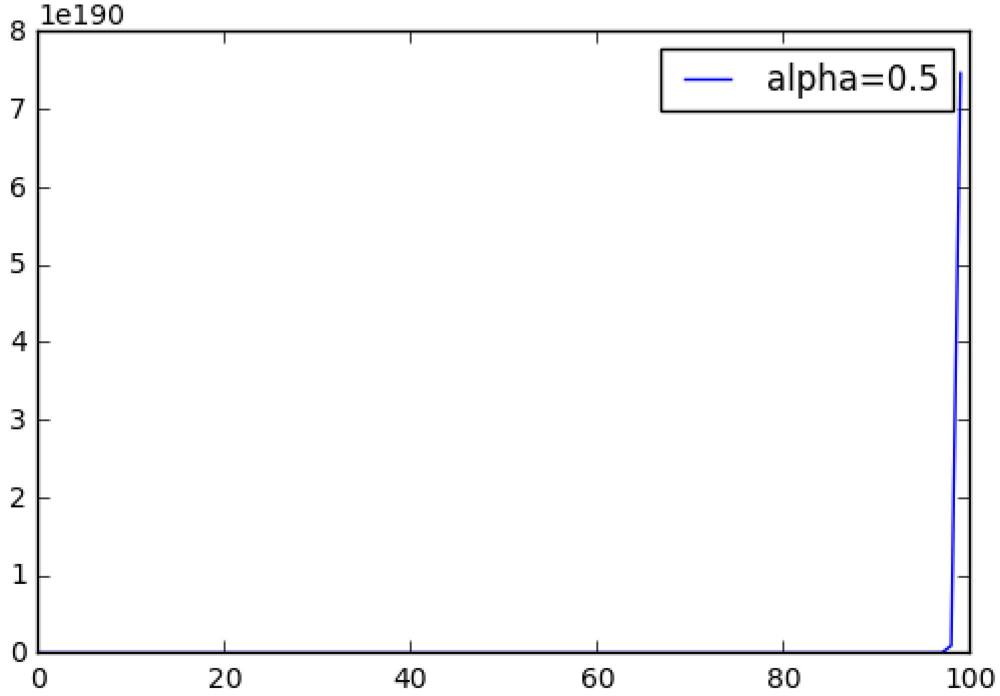
In [14]:

```
import matplotlib.pyplot as plt
a1, b1 = batch_grad_descent(X_train, y_train, alpha=0.5, num_iter=1000, check_gradient=False)
a2, b2 = batch_grad_descent(X_train, y_train, alpha=0.1, num_iter=1000, check_gradient=False)
a3, b3 = batch_grad_descent(X_train, y_train, alpha=0.05, num_iter=1000, check_gradient=False)
a4, b4 = batch_grad_descent(X_train, y_train, alpha=0.01, num_iter=1000, check_gradient=False)
x = list(range(100))
plt.plot(x, b2[0:100])
plt.plot(x, b3[0:100])
plt.plot(x, b4[0:100])
plt.legend(['alpha=0.1', 'alpha=0.05', 'alpha=0.01'], loc='upper right')
plt.show()
```



In [15]:

```
plt.plot(x, b1[0:100])
plt.legend(['alpha=0.5'], loc='upper right')
plt.show()
```




### 2.4.2

From the plot above we can conclude that when alpha is 0.05, it converges the fastest, when alpha is 0.1, it converges the slowest. When alpha is 0.5, it diverges.

# Question 2.5

## 2.5.1

### The gradient of J

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

## Updating $\theta : \theta - step \cdot \nabla J(\theta)$

## 2.5.2

In [16]:

```
#######################################################
### Compute the gradient of Regularized Batch Gradient Descent
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)
    """
    #TODO
    grad = np.zeros(X.shape[1])
    grad = np.dot(X.T,np.dot(X,theta)-y)/X.shape[0] + 2*lambda_reg*theta
    return grad
```

In [17]:

```
compute_regularized_square_loss_gradient(X, y, theta, lambda_reg=0.1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-30ee6700a729> in <module>()
----> 1 compute_regularized_square_loss_gradient(X, y, theta, lambda_reg=0.1)

<ipython-input-16-a2b0db466747> in compute_regularized_square_loss_gradient(X, y, th
eta, lambda_reg)
     16     #TODO
     17     grad = np.zeros(X.shape[1])
---> 18     grad = np.dot(X.T,np.dot(X,theta)-y)/X.shape[0] + 2*lambda_reg*theta
     19     return grad

ValueError: shapes (200,48) and (3,) not aligned: 48 (dim 1) != 3 (dim 0)
```

In [18]:

```
#####################################################
### Batch Gradient Descent with regularization term
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_featur
        loss_hist - the history of regularized loss value, 1D numpy array
    """
    (num_instances, num_features) = X.shape
    theta = np.ones(num_features) #Initialize theta
    theta_hist = np.zeros((num_iter+1, num_features))  #Initialize theta_hist
    loss_hist = np.zeros(num_iter+1) #Initialize loss_hist
    #TODO
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta_hist[0])
    #print(theta_hist[0])
    for i in range(1,num_iter+1):
        theta_hist[i] = theta_hist[i-1]-alpha*compute_regularized_square_loss_gradient(X, y, theta_h
        #print(theta_hist[i])
        loss_hist[i] = compute_square_loss(X, y, theta_hist[i])
    return theta_hist, loss_hist
```

# 2.5.4

### When the B is very big, the corresponding coefficient $\theta_0$ will be very small, which means in the regularization term $\lambda\theta^T\theta$, the bias term's coefficient $\theta_0$ will have a fairly small weight and will be regularized less. So a bigger bias term will decrease the regularization on bias term.

### To increase the regularization, we can make B larger, to decrease, make it smaller.

# 2.5.7

In [19]:

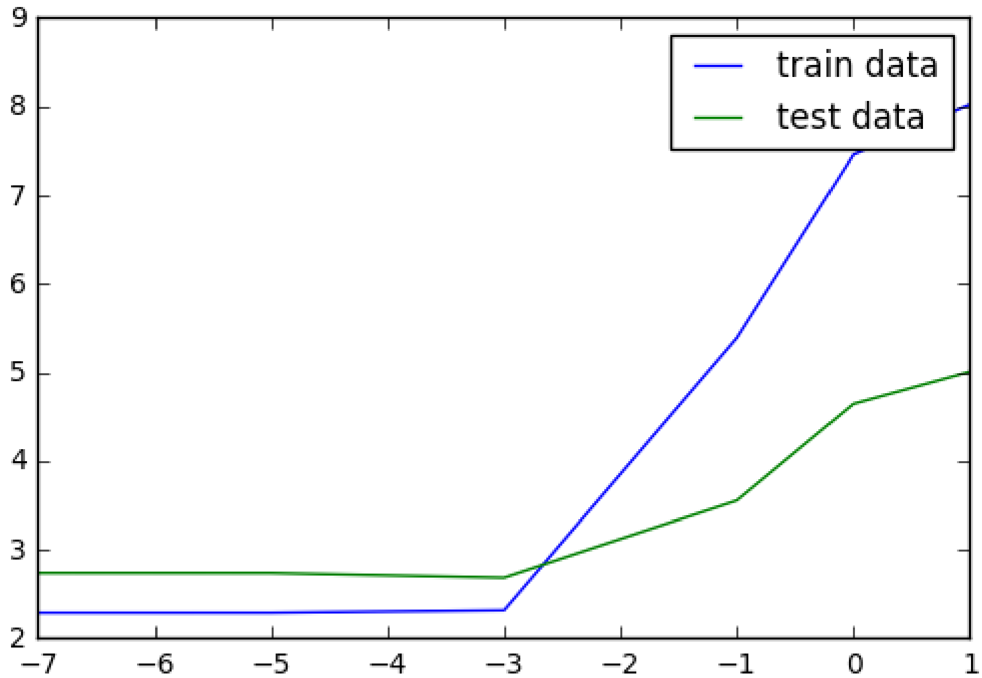
```
# train the model with training data
loss_train=[]
theta_train=[]
labda = [10**(-7),10**(-5),10**(-3),0.1,1,10]
for l in labda:
    theta, loss = regularized_grad_descent(X_train, y_train, alpha=0.05, lambda_reg=1, num_iter=100(
    index = np.argmin(loss)
    loss_train.append(loss[index])
    theta_train.append(theta[index])
```

In [20]:

```
# fit the model with testing data
loss_test=[]
for i in range(len(loss_train)):
    loss_test.append(compute_square_loss(X_test, y_test, theta_train[i]))
```

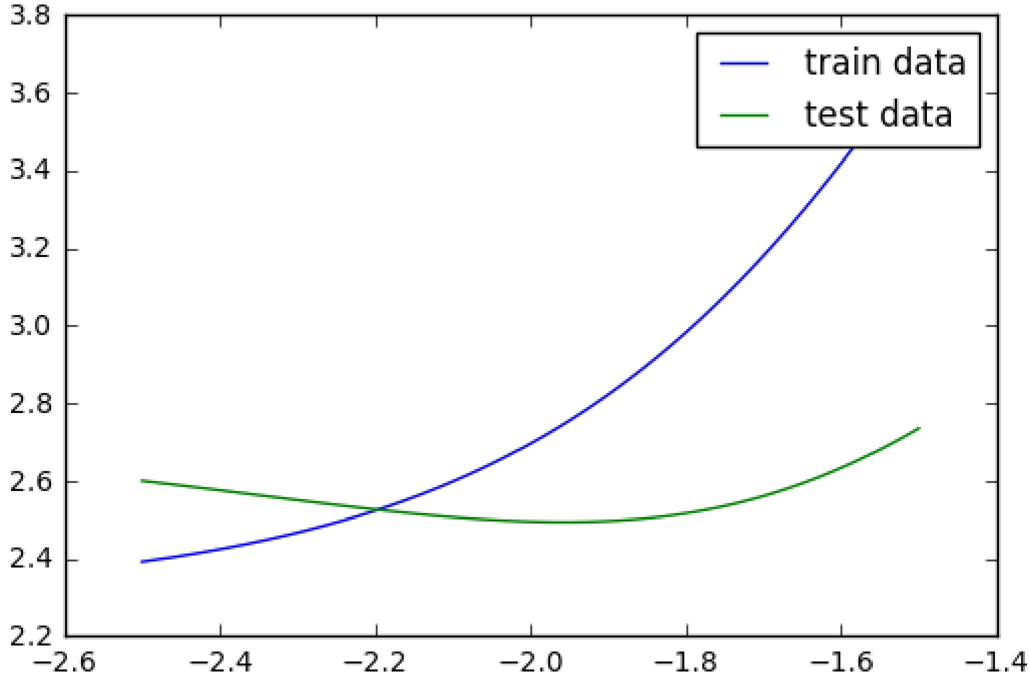
In [21]:

```
# plot the loss line for training data and testing data
x = np.log10(labda)
plt.plot(x,loss_train)
plt.plot(x,loss_test)

plt.legend(['train data','test data'],loc = 'upper right')
plt.show()
```



In [22]:

```
# Zoom in
loss_train=[]
theta_train=[]
loss_test=[]
labda = np.linspace(10**(-2.5),10**(-1.5),100)
for l in labda:
    theta, loss = regularized_grad_descent(X_train, y_train, alpha=0.05, lambda_reg=l, num_iter=1000
    index = np.argmin(loss)
    loss_train.append(loss[index])
    theta_train.append(theta[index])

for i in range(len(loss_train)):
    loss_test.append(compute_square_loss(X_test, y_test, theta_train[i]))
# plot the loss line for training data and testing data
x = np.log10(labda)
plt.plot(x,loss_train)
plt.plot(x,loss_test)

plt.legend(['train data','test data'],loc = 'upper right')
plt.show()
```




## 2.5.8

In [23]:

```
# Find the best theta
print("lambda is")
print(labda[np.argmin(loss_test)])
theta_train[np.argmin(loss_test)]
```

```
lambda is
0.0109242319169
```

Out[23]:

```
array([-1.13486579,  0.48819189,  1.32880538,  2.13057542, -1.59931905,
       -0.76335578, -0.75811346, -0.75811346,  0.66510935,  1.31825504,
        2.1992773 , -0.38022849, -1.32726952, -3.59177059,  1.37342122,
        2.18495616,  1.23304391,  0.44782804, -0.05353292, -0.05353292,
       -0.05353292, -0.02223223, -0.02223223, -0.02223223,  0.00858276,
        0.00858276,  0.00858276,  0.02357895,  0.02357895,  0.02357895,
        0.03212721,  0.03212721,  0.03212721, -0.03246195, -0.03246195,
       -0.03246195,  0.09395789,  0.09395789,  0.09395789,  0.07671455,
        0.07671455,  0.07671455,  0.06890258,  0.06890258,  0.06890258,
        0.06461809,  0.06461809,  0.06461809, -1.20923954])
```

### The result above is the $\theta$ we choose, which is the $\theta$ corresponding to the smallest loss when applied the model with testing data.

# Question 2.6

## 2.6.1

### When using SGD, $J(\theta) = 0.5 \cdot (h_\theta(X_i) - y_i)^2 + \lambda\theta^T * \theta$

$$\nabla J(\theta) = h_\theta(X_i - y_i) + 2\lambda\theta$$

$$= x_i^T \cdot (x_i \cdot \theta - y_i) + 2\lambda\theta$$

### So the updated $\theta$ is

$$\theta = \theta - \eta \cdot \nabla J(\theta) = \theta = \theta - \eta \cdot x_i^T \cdot (x_i \cdot \theta - y_i) + 2\lambda\theta)$$

## 2.6.2

In [24]:

```
#################################################
### Stochastic Gradient Descent
def stochastic_grad_descent(X, y, alpha=0.1, lambda_reg=1, num_iter=1000):
    """
    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
                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_instance
        loss hist - the history of regularized loss function vector, 2D numpy array of size(num_iter
    """
    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

    #TODO
    #shuffle the data
    time_hist = np.zeros(num_iter)
    arr = np.arange(num_instances)
    t=1
    for iteration_index in range(num_iter):
        np.random.shuffle(arr)
        for i in range(num_instances):
            # set the step
            theta_hist[iteration_index][i] = theta
            loss = compute_square_loss(X, y, theta)
            loss_hist[iteration_index][i] = loss
            if isinstance(alpha, float):
                alpha_f = alpha
            elif alpha == '1/sqrt(t)':
                alpha_f = 0.01/(t**(0.5))
            elif alpha == '1/t':
                alpha_f = 0.1*1/(t)
            theta = theta - alpha_f * compute_regularized_square_loss_gradient(X[[arr[i]],:], y[arr[
            t=t+1
    return theta_hist, loss_hist
```

# 2.63

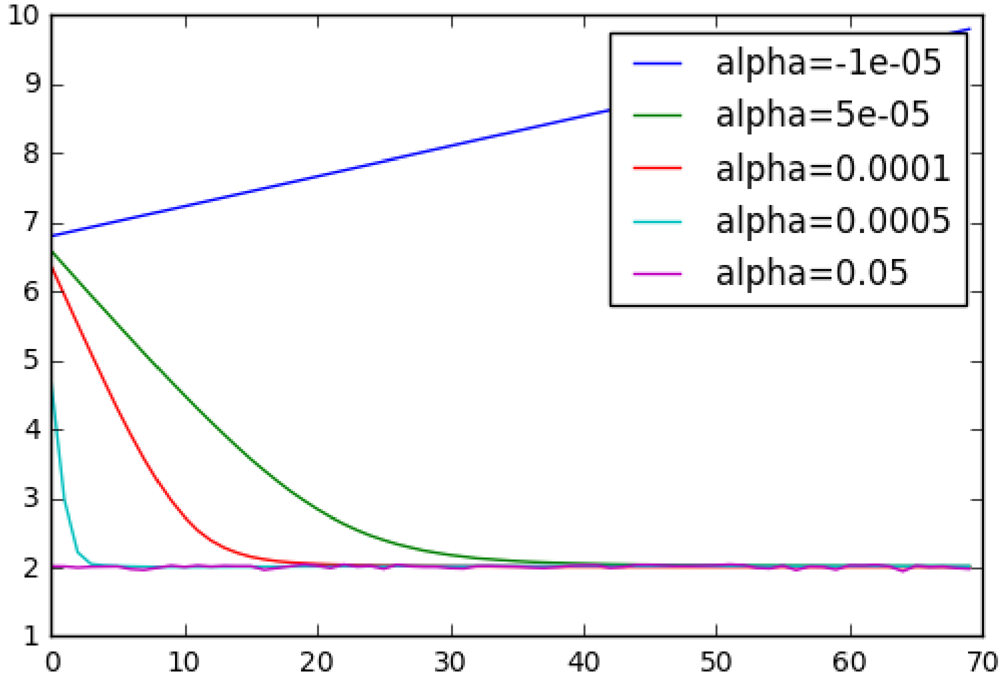
In [43]:

```
def plotSGD(X_train, y_train, alpha, lambda_reg=0.0109242319169, num_iter=70):
    list=[]
    theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, alpha=alpha, lambda_reg=1, num
    print("the best theta for alpha=", str(alpha), " is: ")
    print(theta[np.argmin(loss_hist)/1000][:])
    list = np.amin(loss_hist,axis=1)
    plt.plot(range(num_iter),np.log(list),label="alpha="+str(alpha))
    plt.set_ylim = ([0,7])
    plt.legend(loc = "upper right")

a_list = [-10**(-5),5*10**(-5),10**(-4),5*10**(-4),0.05]
for a in a_list:
    plotSGD(X_train, y_train, alpha=a, lambda_reg=0.0109242319169, num_iter=70)
plt.show()
```

```
the best theta for alpha= -1e-05  is:
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

```
C:\Users\liuyt\Anaconda3\lib\site-packages\ipykernel\__main__.py:5: VisibleDeprecati
onWarning: using a non-integer number instead of an integer will result in an error
 in the future
```

```
the best theta for alpha= 5e-05  is:
[-0.28828415 -0.22377508 -0.17837446 -0.16101414 -0.204626   -0.1870703
 -0.1552813  -0.1552813  -0.09267643 -0.00336543  0.06582126  0.06747731
  0.14207239  0.21217238  0.52817704  0.58840518  0.69942744  0.92971216
 -0.13790456 -0.13790456 -0.13790456 -0.07548706 -0.07548706 -0.07548706
 -0.00375674 -0.00375674 -0.00375674  0.02962012  0.02962012  0.02962012
  0.04818355  0.04818355  0.04818355  0.34815834  0.34815834  0.34815834
  0.24023001  0.24023001  0.24023001  0.17711051  0.17711051  0.17711051
  0.14847514  0.14847514  0.14847514  0.13277113  0.13277113  0.13277113
 -0.30868842]
the best theta for alpha= 0.0001  is:
[-0.29257315 -0.23607951 -0.19451913 -0.17763665 -0.20796805 -0.18962447
 -0.15557115 -0.15557115 -0.09557312 -0.00600496  0.06601172  0.07776648
  0.16524842  0.24181837  0.53780475  0.59480807  0.70511832  0.93378862
 -0.14354022 -0.14354022 -0.14354022 -0.07733872 -0.07733872 -0.07733872
 -0.00408635 -0.00408635 -0.00408635  0.02998016  0.02998016  0.02998016
  0.0489213   0.0489213   0.0489213   0.35756271  0.35756271  0.35756271
  0.24583572  0.24583572  0.24583572  0.18102076  0.18102076  0.18102076
  0.15161003  0.15161003  0.15161003  0.13547889  0.13547889  0.13547889
 -0.31059231]
the best theta for alpha= 0.0005  is:
[-0.30666888 -0.28232553 -0.25568941 -0.24020301 -0.21543571 -0.19310789
 -0.14859393 -0.14859393 -0.09817534 -0.00600752  0.07862941  0.13175047
  0.27182392  0.37393766  0.58631758  0.62955969  0.73466744  0.95145246
 -0.16053333 -0.16053333 -0.16053333 -0.07757852 -0.07757852 -0.07757852
  0.00197565  0.00197565  0.00197565  0.03888999  0.03888999  0.03888999
  0.05938841  0.05938841  0.05938841  0.40422292  0.40422292  0.40422292
  0.27689499  0.27689499  0.27689499  0.20487262  0.20487262  0.20487262
  0.17216091  0.17216091  0.17216091  0.1542098   0.1542098   0.1542098
 -0.31504635]
the best theta for alpha= 0.05  is:
[-0.28828415 -0.22377508 -0.17837446 -0.16101414 -0.204626   -0.1870703
 -0.1552813  -0.1552813  -0.09267643 -0.00336543  0.06582126  0.06747731
  0.14207239  0.21217238  0.52817704  0.58840518  0.69942744  0.92971216
 -0.13790456 -0.13790456 -0.13790456 -0.07548706 -0.07548706 -0.07548706
 -0.00375674 -0.00375674 -0.00375674  0.02962012  0.02962012  0.02962012
  0.04818355  0.04818355  0.04818355  0.34815834  0.34815834  0.34815834
  0.24023001  0.24023001  0.24023001  0.17711051  0.17711051  0.17711051
  0.14847514  0.14847514  0.14847514  0.13277113  0.13277113  0.13277113
 -0.30868842]
```

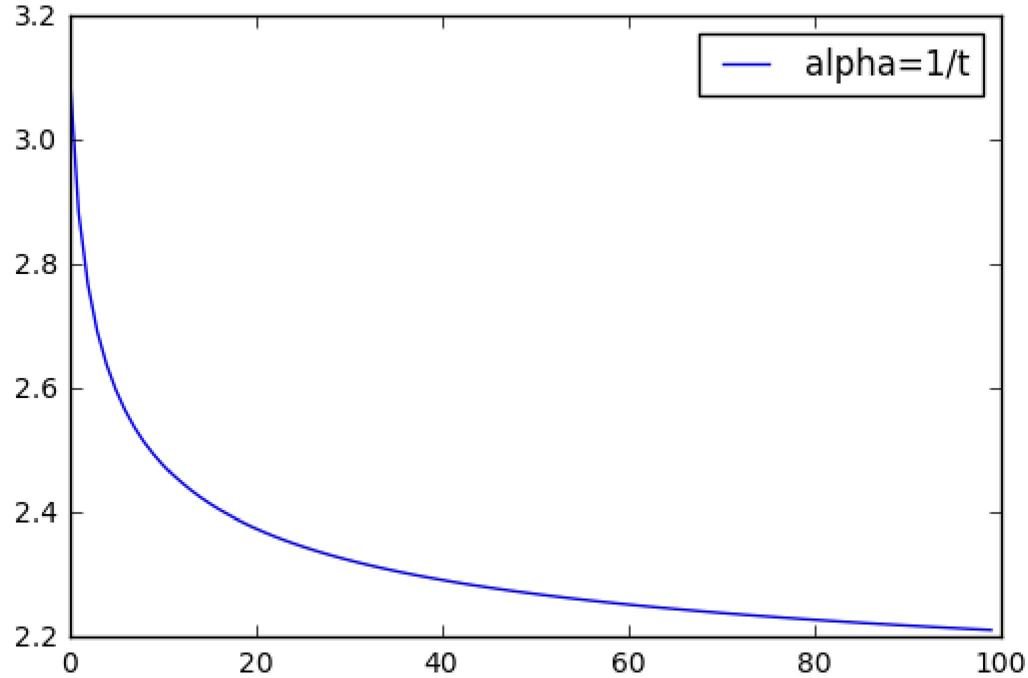



**It's clear that when using fixed steps, incresae the value of alpha will speed up the converging process, however, when the alpha is too big or alpha is less than 0, like alpha = 0.05, -0.00001 in the plot, it will diverge.**

In [37]:

```
plotSGD(X_train, y_train, alpha='1/t', lambda_reg=0.0109242319169, num_iter=100)
plt.show()
```

```
the best theta for alpha= 1/t  is:
[-0.27883536 -0.19032998 -0.13314161 -0.11403601 -0.1962912  -0.18058206
 -0.15468407 -0.15468407 -0.08365182  0.00605239  0.06778039  0.03993475
  0.07708471  0.1280508   0.50294127  0.57253938  0.68487716  0.91802577
 -0.12265469 -0.12265469 -0.12265469 -0.07040784 -0.07040784 -0.07040784
 -0.00281553 -0.00281553 -0.00281553  0.02868586  0.02868586  0.02868586
  0.04622196  0.04622196  0.04622196  0.32238242  0.32238242  0.32238242
  0.22511236  0.22511236  0.22511236  0.16657134  0.16657134  0.16657134
  0.14002824  0.14002824  0.14002824  0.12547631  0.12547631  0.12547631
 -0.3063401 ]

C:\Users\liuyt\Anaconda3\lib\site-packages\ipykernel\__main__.py:5: VisibleDeprecati
onWarning: using a non-integer number instead of an integer will result in an error
 in the future
```



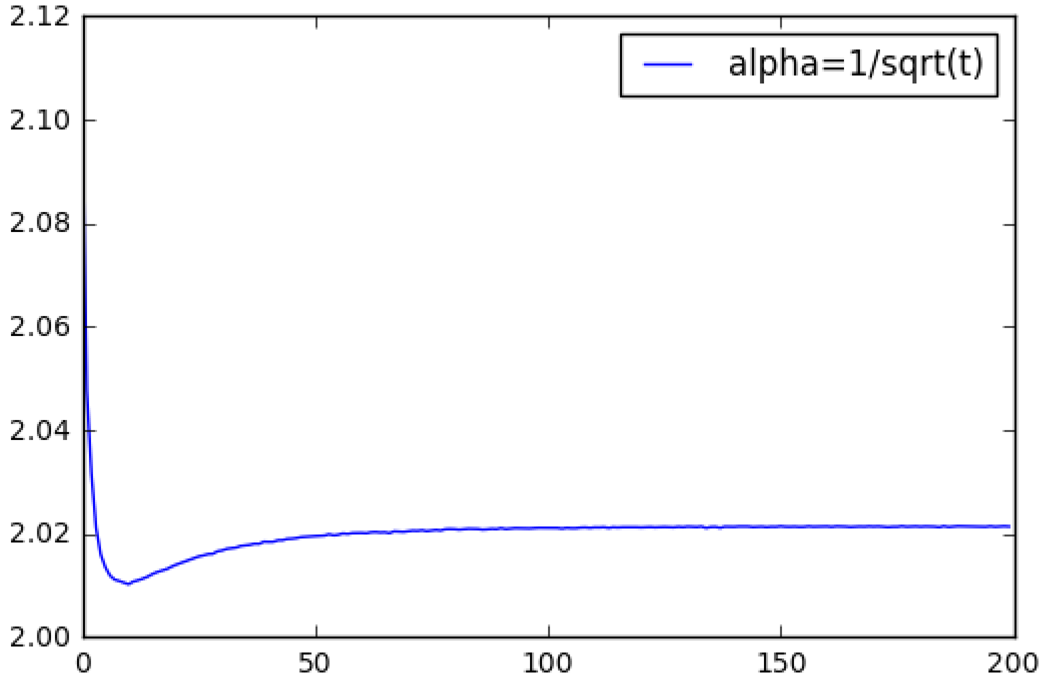

When using 1/sqrt(t), it's also converges, the speed is slower than 1/t

In [45]:

```
plotSGD(X_train, y_train, alpha='1/sqrt(t)', lambda_reg=0.0109242319169, num_iter=200)
plt.show()
```

```
the best theta for alpha= 1/sqrt(t)  is:
[-0.30666888 -0.28232553 -0.25568941 -0.24020301 -0.21543571 -0.19310789
 -0.14859393 -0.14859393 -0.09817534 -0.00600752  0.07862941  0.13175047
  0.27182392  0.37393766  0.58631758  0.62955969  0.73466744  0.95145246
 -0.16053333 -0.16053333 -0.16053333 -0.07757852 -0.07757852 -0.07757852
  0.00197565  0.00197565  0.00197565  0.03888999  0.03888999  0.03888999
  0.05938841  0.05938841  0.05938841  0.40422292  0.40422292  0.40422292
  0.27689499  0.27689499  0.27689499  0.20487262  0.20487262  0.20487262
  0.17216091  0.17216091  0.17216091  0.1542098   0.1542098   0.1542098
 -0.31504635]
```

```
C:\Users\liuyt\Anaconda3\lib\site-packages\ipykernel\__main__.py:5: VisibleDeprecati
onWarning: using a non-integer number instead of an integer will result in an error
 in the future
```




When using 1/t, it's also converges, the speed is faster than 1/sqrt(t)

## 2.64

In [265]:

```
import timeit
import time
def cal_time(X, y, alpha=0.1, lambda_reg=1, num_iter=1000):
    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
    #TODO
    time_hist = np.zeros(num_iter)
    index = np.arange(num_instances)
    shuffled_index = np.random.shuffle(index)
    for iteration_index in range(num_iter):
        np.random.shuffle(arr)
        start = time.time()
        for i in range(num_instances):
            # set the step
            if isinstance(alpha, float):
                alpha_f = alpha
            elif alpha == "1/sqrt(t)":
                alpha = 1/sqrt(i)
            elif alpha == "1/t":
                alpha = 1/i
            theta_hist[iteration_index][i] = theta
            loss = compute_square_loss(X, y, theta)
            loss_hist[iteration_index][i] = loss
            theta = theta - alpha_f * compute_regularized_square_loss_gradient(X[[index[i]],:], y[ir

        stop=time.time()
        time_hist[iteration_index] = stop-start

    print(float(sum(time_hist)/len(time_hist)))
    return
```

In [449]:

```
# calculate the average times for each step.
cal_time(X_train, y_train, alpha=0.1, lambda_reg=1, num_iter=1000)
```

0.003946572542190552

## 2.6.5

### When we want to minimize the optimization time, we will use SGD, when we want to minimize the numbers of epoches or steps, we will chose batch gradient descent. Since SGD only use one sample to approximate the gradient, compared to taking all the instances when using gradient descent method, SGD will be faster, but need more times of iterations though.

# Question 3

## 3.1

**(i)** $E(a-y)^2$
$= E[(a-E(y))^2 + (E(y))^2 + 2(a-E(y)(E(y)-y))$
$= E(a-E(y))^2 + E(E(y)-y)^2 + 2E(a-E(y))(E(y)-y))$

**The last term is 0 and $E(E(y)-y)^2$ is greater or equal than 0. So** $argmin_a E(a-y)^2 = E(y)$

**(ii)**

**When $a = E(y)$**

$E(a-y)^2$
$= E(E(y)-E(y))^2 + E(E(y)-y)^2$
$= E(E(y)-y)^2$
$= E(y^2) - (E(y))^2$
$= Var(y)$

## 3.2

**(a)**

$R(f) = E(f(x)-y)^2$
$= E[E(f(x)-y)^2|x]$
$= E[E(f(x)-E[y|x]+E[y|x]-y)^2|x]$
$= E[E(f(x)-E(y|x)^2)|x + E(E(y|x)-y)^2|x] + 2E(E(f(x)-E(y|x))(E[y|x]-y)|x)$
$= E(f(x)-E[y|x]^2) + E(E(y|x)-y)^2$

**So , R(f) is minimized when f=E[y|x], So f*=E[y|x]**

**(b)**

$R(f) = E(f(x)-E[y|x])^2 + E[E[y|x]-y]^2$

**So**

$R(f = f*) = E(E[y|x] - E[y|x])^2 + E(E[y|x] - y)^2$
$= E(E[y|x] - y)^2$

**So**

$R(f) = E(f(x) - E[y|x])^2 + R(f*)$

**So**

$R(f*) \leq R(f)$ **as** $R(f) = E(f(x) - y)^2$

**So**

$E[(f*(x) - y)^2] \leq E(f(x) - y)^2$