In [1]:
```
import pandas as pd
import logging
import numpy as np
import sys
import matplotlib.pyplot as plt
import time
from sklearn.cross_validation import train_test_split
%pylab inline
```

```
Populating the interactive namespace from numpy an
d matplotlib
```

In [4]:
```
def init():
    #Loading the dataset
    print('loading the dataset')

    df = pd.read_csv('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_spli
t(X, y, test_size =100, random_state=10)
    return X_train, X_test, y_train, y_test
#!---init---!
X_train, X_test, y_train, y_test = init()
```

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

2. Linear Regression

2.1 Feature Normalization

Modify function `feature_normalization` to normalize all the features to [0,1].

Answer:

Min-Max Scaling is used to normalize all the features.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

When subtracting vector X_{min} from matrix X, and dividing $X_{max} - X_{min}$, Numpy's "broadcasting" is used.

In [5]:

```
#######################################
####Q2.1: Normalization
def feature_normalization(train, test):
    """Rescale the data so that each feature in the tr
aining set is in
    the interval [0,1], and apply the same transformat
ions 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 (nu
m_instances, num_features)
    Returns:
        train_normalized - training set after normaliz
ation
        test_normalized  - test set after normalizatio
n

    """
    # Min-Max Scaling
    train_min = np.min(train,axis=0)
    train_max = np.max(train,axis=0)
    return (train-train_min)/(train_max - train_min+0.
0), (test-train_min)/(train_max - train_min+0.0)

#!---Scale---!

print("Scaling all to [0, 1]")
X_train, X_test = feature_normalization(X_train, X_tes
t)
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
```

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

2.2 Gradient Descent Setup

1.Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign.

ANSWER:

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

2.Write down an expression for the gradient of J.

ANSWER:

$$\nabla_\theta J(\theta)$$

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

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

3.Use the gradient to write down an approximate expression for $J(\theta + \eta\Delta) - J(\theta)$

ANSWER:

The gradient at point θ is the best linear approximation of J at that point.

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

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

ANSWER:

$$\theta_{i+1} = \theta_i - \eta\nabla_\theta J$$

5.Modify the function `compute_square_loss`, to compute $J(\theta)$ for a given θ.

In [6]:
```
#########################################
####Q2.2a: The square loss function

def compute_square_loss(X, y, theta):
    """
    Given a set of X, y, theta, compute the square los
s 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 = np.dot(X, theta) - y
    return 0.5 * np.sum(loss ** 2) / X.shape[0]
```

6.Create a small dataset for which you can compute $J(\theta)$ by hand, and verify that your `compute_square_loss` function returns the correct value.

In [7]:
```
theta = np.random.rand(X_train.shape[1])
print compute_square_loss(X_train, y_train, theta)
```

```
115.663742555
```

7.Modify the function `compute_square_loss_gradient`, to compute $\nabla_\theta J(\theta)$.

In [8]:
```
###Q2.2b: 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)
    """
    m = X.shape[0]
    loss = np.dot(X, theta)-y
    return 1/(m+0.0)*np.dot(X.T, loss)
```

8.Create a small dataset, verify that your `compute_square_loss_gradient` function returns the correct value.

In [9]:
```
theta = np.random.rand(X_train.shape[1],1)
#print compute_square_loss_gradient(X_train, y_train, theta)
```

In [10]:
```
X_new = np.random.rand(3,3)
y_new = np.random.rand(3,1)
theta_new = np.random.rand(3,1)
print compute_square_loss_gradient(X_new, y_new, theta
_new)
```

```
[[ 0.11544333]
 [ 0.10037056]
 [-0.02513624]]
```

2.3 Gradient Checker

In [11]:

```
###Q2.3a: 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

    for i in range(num_features):
        e_i = np.zeros(num_features)
        e_i[i] = 1
        approx_grad[i] = (compute_square_loss(X, y, theta + epsilon *e_i) -
                          compute_square_loss(X, y, theta + epsilon *e_i))/(2*tolerance+0.0)
    dist = np.sqrt(np.sum((approx_grad-true_gradient))**2)
    correct_grad = dist<tolerance
    assert correct_grad, "Gradient bad: dist %s is gre
```

```
ater than tolerance %s" % (dist,tolerance)
    return correct_grad
```

2.Write a generic version of `grad_checker` that will work for any objective function. It should take as parameters a function that computes the gradient of the pbjective function.

In [12]:

```
##############################################
###Q2.3b: Generic Gradient Checker
def generic_gradient_checker(X, y, theta, objective_fu
nc, gradient_func, epsilon=0.01, tolerance=1e-4):
    """
    The functions takes objective_func and gradient_fu
nc 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_lo
ss, and gradient_func = compute_square_loss_gradient
    """
    true_gradient = gradient_func(X, y, theta) #the tr
ue gradient
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize t
he gradient we approximate

    for i in range(num_features):
        e_i = np.zeros(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*tolerance+0.0)
    dist = np.sqrt(np.sum((approx_grad-true_gradient))
**2)
    correct_grad = dist<tolerance
    assert correct_grad, "Gradient bad: dist %s is gre
ater than tolerance %s" % (dist,tolerance)
    return correct_grad
```

2.4 Batch Gradient Descent

1.Complete `batch_gradient_descent`

In [15]:

```
#####################################
####Q2.4a: Batch Gradient Descent
def batch_grad_descent(X, y, alpha=0.01, num_iter=1000, check_gradient=False):
    """
    In this question you will implement batch gradient descent to
    minimize the square loss objective

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

    Returns:
        theta_hist - store the the history of parameter vector in iteration, 2D numpy array of size (num_iter+1, num_features)
                    for instance, theta in iteration 0 should be theta_hist[0], theta in ieration (num_iter) is theta_hist[-1]
        loss_hist - the history of objective function vector, 1D numpy array of size (num_iter+1)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) #initialize theta
    theta_hist = theta #Initialize theta_hist
    loss_hist = compute_square_loss(X, y, theta.T) #initialize loss_hist
    for i in range(num_iter):
```

```
        if check_gradient:
            grad = grad_checker(X, y, theta)
            print('Grade check:{0}'.format(grad))
        grad = compute_square_loss_gradient(X,y,theta.
T)
        theta = theta - alpha*grad.T
        theta_hist = np.vstack((theta_hist, theta))
        loss = compute_square_loss(X, y, theta.T)
        loss_hist = np.vstack((loss_hist, loss))

    return loss_hist,theta_hist
```

In [16]:
```
X = X_train
y = y_train
loss_return, theta_return = batch_grad_descent(X, y)
loss_return
```

Out[16]:
```
array([[ 441.43426102],
       [ 285.54947422],
       [ 185.24897342],
       ...,
       [   1.86585135],
       [   1.86503146],
       [   1.86421264]])
```

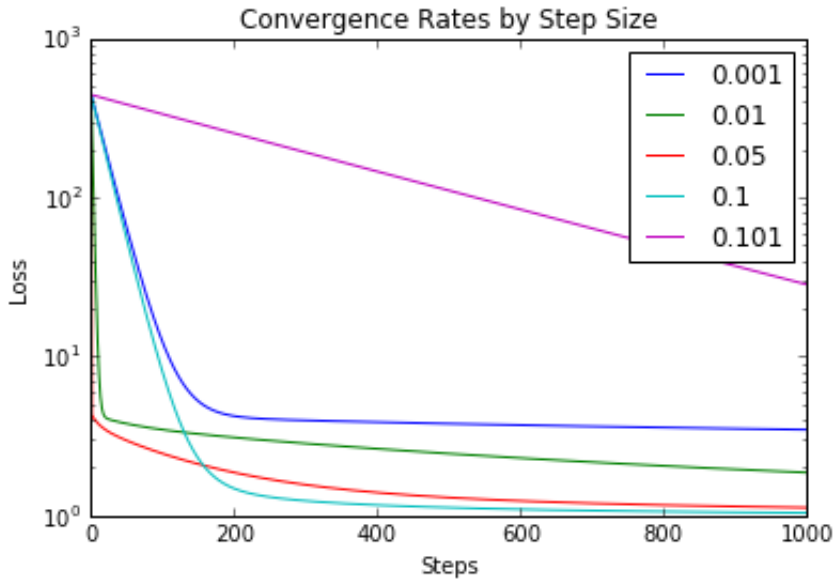
2.Try step sizes 0.5, 0.1, 0.05, 0.01. Plot the value of the objective function as a function of the number of steps for each step sizes. Briefly summarize your findings.

In [17]:

```
num_iter = 1000
def converge_test(X, y):
    step_sizes = np.array([0.001, 0.01, 0.05, 0.1, 0.1
01])
    for step_size in step_sizes:
        loss_hist,_ = batch_grad_descent(X,y,alpha=ste
p_size, num_iter=num_iter)
        plt.plot(loss_hist, label=step_size)

    plt.xlabel('Steps')
    plt.ylabel('Loss')
    plt.yscale('log')
    plt.title('Convergence Rates by Step Size')
    plt.legend()
    plt.show()

converge_test(X_train,y_train)
```



3.(Option)backtracking line search

In [18]:
```
#####################################
###Q2.4b: Implement backtracking line search in batch_
gradient_descent
###Check http://en.wikipedia.org/wiki/Backtracking_lin
e_search for details
#TODO
```

2.5 Ridge Regression

1.Compute the gradient of $J(\theta)$ and write down the expression for updating θ in the gradient descent algprithm

ANSWER:

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

2.Implement `compute_regularized_square_loss_gradient`

In [19]:
```
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg=1):
    """
    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)
    """
    regularization_term = 2.0 * lambda_reg * theta

    grad = compute_square_loss_gradient(X, y, theta) + regularization_term
    return grad
```

3.Implement `regularized_grad_descent`

In [20]:

```
#################################################
###Q2.5b: 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_features)
        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

    for i in range(num_iter + 1):
        loss_hist[i] = compute_square_loss(X, y, theta) + lambda_reg * np.sum(theta ** 2)
        theta_hist[i] = theta

        grad = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)
        #theta = theta - alpha * grad/np.linalg.norm(g
```

```
rad)
        theta = theta - alpha * grad

    return loss_hist,theta_hist
```

In [21]:
```
loss, theta = regularized_grad_descent(X_train, y_trai
n, 0.01, 1)
loss
```

Out[21]:
```
array([ 490.43426102,  303.0924192 ,  188.4087873
8, ...,    3.82298295,
          3.82298295,    3.82298295])
```

4.Explain why making B large decreases the effective regulatization on the bias term, and how we can make that regularization as weak as we like(though not zero)

ANSWER:

The gradient descent algorithm seeks to minimize the loss function by driving the coeffecient of the linear function, i.e. $B \to \infty, \theta_{bias} \to 0$. So using a large B will decrease the impact of regulation λ on the coeffecient of the bias term.

5.Choose a reasonable step size. Plot the training loss and the validation loss(without the regulation) as a function of λ.

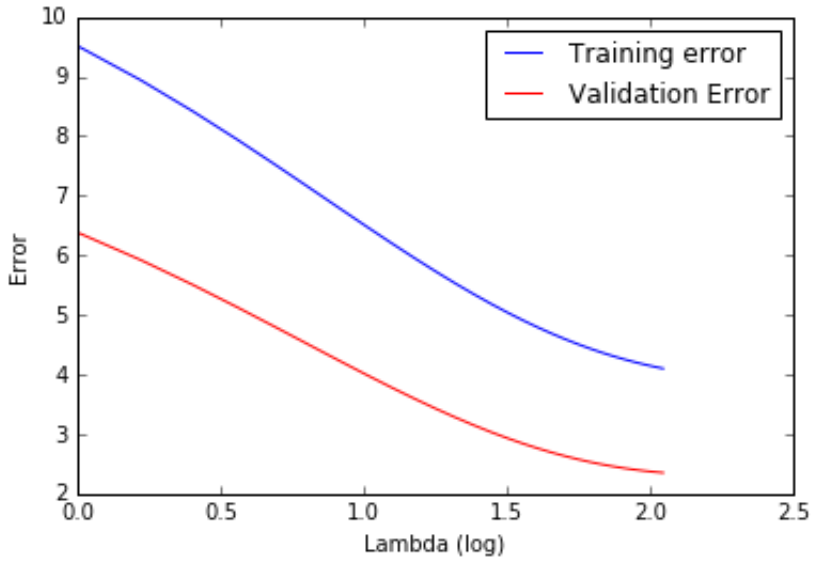
In [26]:

```
##########################################
##Q2.5c: Visualization of Regularized Batch Gradient D
escent
##X-axis: log(lambda_reg)
##Y-axis: square_loss
def vis_regularized_batch_gradient_descent(X_train, X_
test, y_train, y_test, lambdas):
    stepSize = 0.00001
    train_error = np.zeros(lambdas.shape[0])
    validation_error = np.zeros(lambdas.shape[0])
    for i, lamb in enumerate(lambdas):
        losses, thetas = regularized_grad_descent(X_tr
ain, y_train, stepSize, lambda_reg=lamb, num_iter=1000
0)
        train_error[i] = compute_square_loss(X_train,
y_train, thetas[-1,:])
        validation_error[i] = compute_square_loss(X_te
st, y_test, thetas[-1,:])

    print "Min lambda: " + str(lambdas[np.argmin(valid
ation_error)])
    print "Min training error: " + str(train_error.min
())
    print "Min validation error: " + str(validation_er
ror.min())
    plt.plot(np.log(lambdas), train_error,label="Train
ing error", c='b')
    plt.plot(np.log(lambdas), validation_error,label
="Validation Error" ,c='r')
    plt.legend()
    plt.xlabel("Lambda (log)")
    plt.ylabel("Error")
    plt.show()
```

In [27]:
```
#lambdas = np.array([1e-7, 1e-5, 1e-3, 1e-1, 1, 10, 10
0,1000])
lambdas = np.arange(1, 8 , 0.25)
vis_regularized_batch_gradient_descent(X_train, X_tes
t, y_train, y_test, lambdas)
```

```
Min lambda: 7.75
Min training error: 4.09907511755
Min validation error: 2.35093006421
```



2.6 Stochastic Gradient Descent

1.Write down the update rule for θ in SGD.

ANSWER:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \frac{1}{2}(h_\theta(x_i) - y_i)$$

where (x_i, y_i) is randomly chosen porint.

2.Implement `stochastic_grad_descent`.

In [52]:

```
#############################################
# ##Q2.6a: Stochastic Gradient Descent
def stochastic_grad_descent(X, y, alpha=0.001, lambda_
reg=1, num_iter=1000):
    """
    In this question you will implement stochastic gra
dient 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 i
dea to use a fixed step size. Usually it's set to 1/sq
rt(t) or 1/t
                if alpha is a float, then the step siz
e in every iteration is alpha.
                if alpha == "1/sqrt(t)", alpha = 1/sqr
t(t)
                if alpha == "1/t", alpha = 1/t
        lambda_reg - the regularization coefficient
        num_iter - number of epochs (i.e number of tim
es) to go through the whole training set

    Returns:
        theta_hist - the history of parameter vector,
3D numpy array of size (num_iter, num_instances, num_f
eatures)
        loss hist - the history of regularized loss fu
nction vector, 2D numpy array of size(num_iter, num_in
stances)
    """
    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, nu
m_features))  # Initialize theta_hist
    loss_hist = np.zeros((num_iter, num_instances))  #
Initialize loss_hist
    # TODO
    if isinstance(alpha, str):
        if alpha == '1/t':
            f = lambda x: 1.0 / x
        elif alpha == '1/sqrt(t)':
            f = lambda x: 1.0 / np.sqrt(x)
        alpha = 0.01
    elif isinstance(alpha, float):
        f = lambda x: 1
    else:
        return

    t0 = time.time()

    for t in range(num_iter):


        for i in range(num_instances):
            gamma_t = alpha * f((i+1)*(t+1))

            theta_hist[t , i] = theta
            # compute loss for current theta
            loss = np.dot(X[i], theta) - y[i]
            # reg. term
            regulariztion_loss = lambda_reg * np.dot(t
heta.T,theta)
            # squared loss
            loss_hist[t, i] = (0.5) * (loss) ** 2 + re
```

```
gulariztion_loss

            regularization_penalty = 2.0 * lambda_reg
* theta
            grad = X[i] * (loss) + regularization_pena
lty
            theta = theta - gamma_t * grad

    t1 = time.time()
    print "Average time per epoch:: " + str((t1 - t0)
/ num_iter)
    return theta_hist, loss_hist
```

3.Use SDG to find θ^*_λ that minimizes the ridge regression objective for the λ and B that you selected in the previous problem. Try a few fixed step sizes (at least try $\eta_t \in 0.05, .005$). Note that SGD may not converge with fixed step size. Simply note your results. Next try step sizes that decrease with thes step number according to the following schedules: $\eta_t = \frac{1}{t}$ and $\eta_t = \frac{1}{\sqrt{t}}$. For each step size rule, plot the value of the objective function as a function of epoch for each of the approaches to step size. How do the results compare?

In [53]:

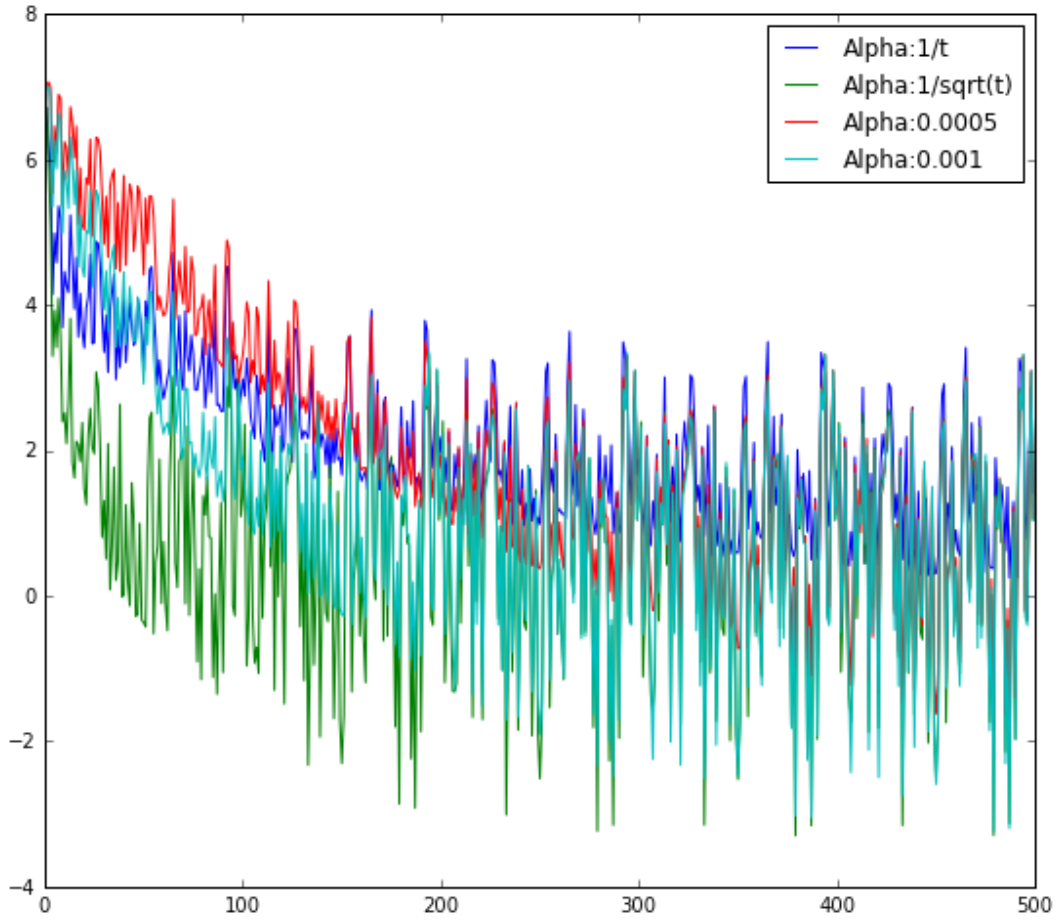
```
def convergence_tests_batch_vs_stochastic(X,y):
    alphas = ['1/t','1/sqrt(t)',0.0005,0.001]
    fig = plt.figure(figsize=(20,8))
    plt.subplot(121)

    for alpha in alphas:
        [thetas,losses] = stochastic_grad_descent(X,
y, alpha, 5.67, 5)
        plt.plot(np.log(losses.ravel()),label='Alph
a:'+str(alpha))

    plt.legend()
```

In [54]:
```
convergence_tests_batch_vs_stochastic(X_train,y_train)
```

```
Average time per epoch:: 0.000810813903809
Average time per epoch:: 0.000934219360352
Average time per epoch:: 0.00155262947083
Average time per epoch:: 0.00154900550842
```



In [ ]:
```
#################################################
###Q2.6b Visualization that compares the convergence s
peed of batch
###and stochastic gradient descent for various approac
hes to step_size
##X-axis: Step number (for gradient descent) or Epoch
(for SGD)
##Y-axis: log(objective_function_value)
```

3.Risk Minimization

1.Show that for the square loss $\ell(\hat{y} - y) = \frac{1}{2}(y - \hat{y})^2$, the Bayes decision function is a $f^*(x) = \mathbb{E}[Y|X = x]$

$R(f)$

$= \frac{1}{2}\mathbb{E}[(f(x) - y)^2]$

$= \frac{1}{2}\mathbb{E}[(y - \hat{y})^2|x]$

$= \frac{1}{2}\mathbb{E}[(y^2 - 2y\hat{y} + \hat{y}^2)|x]$

$= \frac{1}{2}(\mathbb{E}[y^2|x] - 2\mathbb{E}[y|x] + \hat{y}^2)$

Hence,

$\frac{\partial R(f)}{\partial \hat{y}} = 2\hat{y} - 2\mathbb{E}[y|x]$

$\hat{y} = \mathbb{E}[y|x]$

2.Show that for the absolute loss $\ell(\hat{y}, y) = |y - \hat{y}|$, the Bayes function is a $f^*(x) = median[Y|X = x]$.

$R(f)$

$= \mathbb{E}[|y - \hat{y}||x]$

$= \int |y - \hat{y}|\pi(y|x)dy$

$= \int_{y \le \hat{y}} (\hat{y} - y)\pi(y|x)dy + \int_{y \ge \hat{y}} (y - \hat{y})\pi(y|x)dy$

$$\frac{\partial R(f)}{\partial \hat{y}}$$

$$= -\int_{y \geq \hat{y}} \pi(y|x)dy + (\hat{y} - \hat{y})(1) - (\hat{y} - y^{+})(0) + \int_{y \leq \hat{y}} \pi(y|x$$

$$= -\int_{y \geq \hat{y}} \pi(y|x)dy + \int_{y \leq \hat{y}} \pi(y|x)dy = 0$$

Hence,

$$P(y \geq \hat{y}) = P(y \leq \hat{y})$$