Homework 1 - Alex Pine - 2015/02/01

2.1 Feature Normalization

See the feature_normalization function in hw1.py.

2.2 Gradient Descent Setup

1. Express $J(\theta)$ in vection notation.

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

2. Expression for the gradient of J.

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

3. Expression for $J(\theta + \eta\Delta) - J(\theta)$

$$\text{Since } \nabla J(\theta) \approx \frac{J(\theta + \eta\Delta) - J(\theta)}{\eta\Delta}$$
$$J(\theta + \eta\Delta) - J(\theta) \approx \nabla J(\theta)\eta\Delta$$

4. Update expression for θ

$$\theta = \theta - \eta[\frac{1}{m}\frac{(X\theta - Y)^T X}{||(X\theta - Y)^T X||}]$$

5. Compute the gradient of square loss function

See the compute_square_loss_gradient function in hw1.py

2.3 Gradient Checker

See the grad_checker and generic_gradient_checker functions in hw1.py

2.4 Batch Gradient Descent

1. batch_gradient_descent

See the batch_gradient_descent function in hw1.py

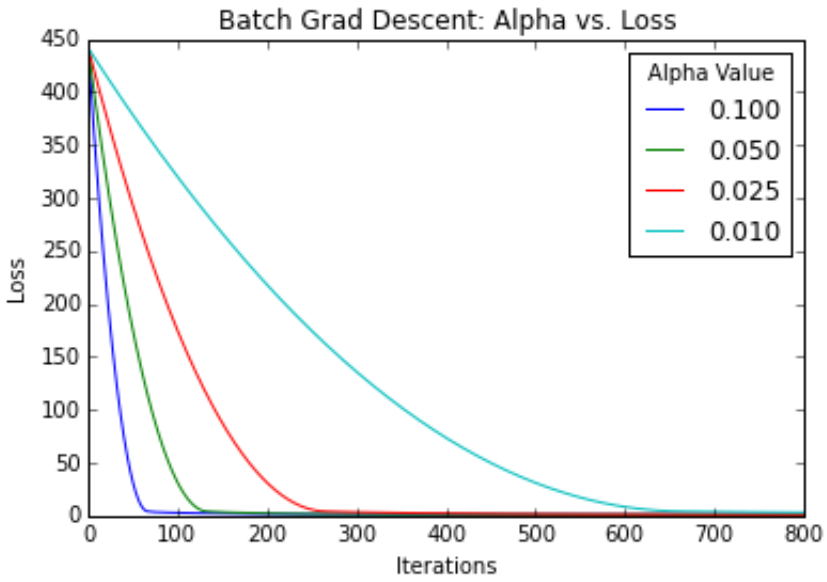
2. Graph convergence rate against step size

If alpha was much greater than 0.1, two things happened: first, the steps were too big for it ever to converge. Second, the gradient could not be computed accurately, so it wasn't moving towards the minimum. The smaller step sizes converged quickly. One could imagine that, in a larger data set, the smaller steps would require more iterations to converge.

In [59]:
```
%matplotlib inline

import sys
sys.path.append('/Users/pinesol/machine-learning/hw1-sgd/')
import hw1
reload(hw1)
alphas = [0.1, 0.05, 0.025, 0.01]
num_iterations = 800
hw1.plotBatchGradientDescentConvergence(alphas, num_iterations)
```

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



2.5 Ridge Regression

1. Compute the gradient of J(θ) and write down the expression for updating θ in the gradient descent algorithm

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

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

2. Implement compute_regularized_square_loss_gradient.

See the compute_regularized_square_loss_gradient function in hw1.py

3. Implement regularized_grad_descent.

See the regularized_grad_descent function in hw1.py

4. How can we avoid regularizing the bias term?

By including the bias term in the design matrix X, we subject it to the shrinking effect of the regularization term $\lambda\theta^T$. The bias term controls the distance of the hyperplane to the origin, and there is no reason we should prefer a hyperplane that is close to the origin. We therefore should not regularize the bias term. This complicates the code, however, so as an alternative, we can make the bias term very large, so as to offset any shrinkage caused by regularization term.

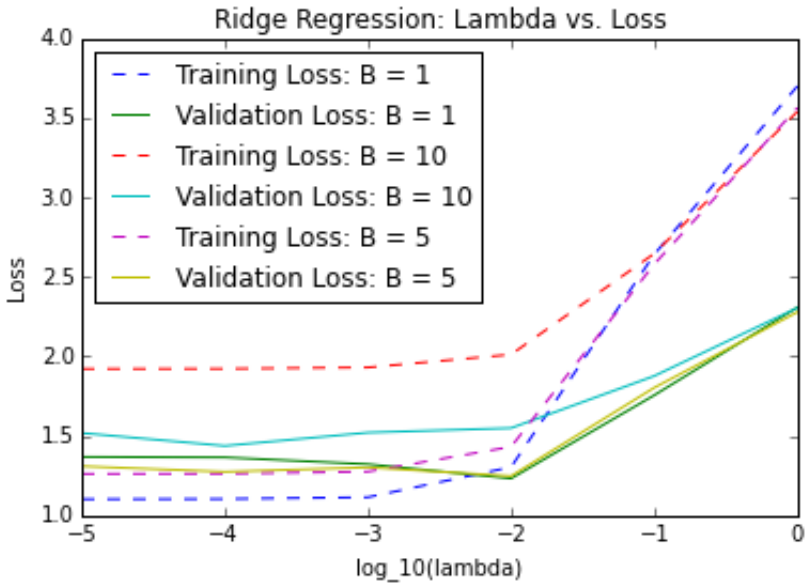
5. Finding a bias weight and regularization weight

I experimented with bias terms of 1, 8, and 16. From the graph below, you can see that changing the bias term changed the loss from the final theta quite a bit. The value of B that resulted in the lowest loss was B = 1, implying regularization did not have a significant effect on the bias.

In [62]:
```
reload(hw1)

alpha = 0.02
num_iterations = 1000
lambda_powers=range(-5, 1) # This is 10^(-5), 10^(-4), ..., 10^1
Bs=[1, 5, 10]
hw1.plotRidgeRegressionLossAgainstLambda(alpha, num_iterations, lambda
_powers, Bs)
```

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



2.6 Ridge Regression

6. Estimate the average time it takes on your computer to compute a single gradient step.

Averaging the amount of time to compute a gradient step using the time.time() function over 500 iterations, I found that the average gradient step required roughly 0.02 milliseconds to compute.

7. What θ and B would you select for deployment and why?

I'm assuming lambda was intended in the question, and not theta.

I would choose lambda to be 0.001, because that is the largest amount of regularization that minimizes the loss. Smaller values of lambda have result in roughly the same amount of loss in this validation set, and would presumably cause greater loss than a larger value.

Different values of B don't seem to make a difference. When they get much larger, they cause the gradient to get so large that the algorithm doesn't converge.

2.6 Stochastic Gradient Descent

1. Write down the update rule for θ in SGD

On the ith iteration of an epoch, the iteration rule is:

$$\theta_{i+1} = \theta_i - \alpha[((x_i \cdot \theta) - y_i)x_i) + 2\lambda\theta]$$

Where x_i is a single row vector of the design matrix X, and y_i is a scalar from the response vector Y.

2. Implement stochastic grad descent.

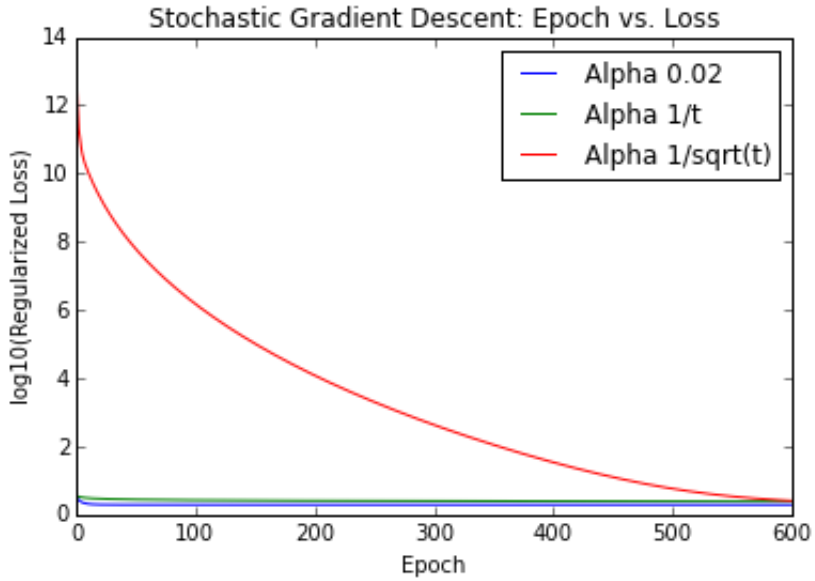
See the stochastic_grad_descent function in hw1.py

3. Find the optimal theta and learning rate function

Surprisingly, a fixed step size, (0.02 was chosen arbitrarily) converged the quickest. $\alpha(t) = \frac{1}{\sqrt{t}}$ converged the slowest, because its step size starts out comparitively large.

In [75]:
```
reload(hw1)
hw1.main(bias=1, lambda_reg=0.01, num_iter=600, alphas=[.02, "1/t", "1
/sqrt(t)"])
```

```
loading the dataset
Split into Train and Test
Scaling all to [0, 1]
Training using stochastic gradient descent with alpha 0.02
Training using stochastic gradient descent with alpha 1/t
Training using stochastic gradient descent with alpha 1/sqrt(t)
```



4. Estimate the amount of time it takes on your computer for a single epoch of SGD

Each epoch averaged roughly 2.7 milliseconds.

5. Compare the speed SGD and batch gradient descent

Each SGD epoch was over ten times slower than the each iteration of batch gradient descent (2.7 millisconds versus 0.2 milliseconds). This is likely due to the garbage collection and memory allocation overheard required for each inner loop of SGD. However, if the dataset were much larger, it's likely SGD would be faster, since only a few epochs are generally required to converge to the minimum.

3. Risk Minimization

3.1 Square Loss

Show that the $f(x)$ that minimizes the risk $R(y,f(x)) = E[\frac{1}{2}(y - f(x))^2]$, is $E[y|X = x]$

Let $\hat{y} = f(x)$, then $R(y, \hat{y}) = E[\frac{1}{2}(y - \hat{y})^2]$, and the smallest $\hat{y}$ can make R occurs when $\frac{dR}{d\hat{y}} = 0$

$$R(y,\hat{y}) = E[\frac{1}{2}(y-\hat{y})^2] = \frac{1}{2}\int (y-\hat{y})^2)p(y|X=x)dy$$

$$R(y,\hat{y}) = \frac{1}{2}[\int y^2 p(y|X=x)dy - 2\hat{y}\int yp(y|X=x)dy + \hat{y}^2 \int p(y|X=x)dy]$$

$$R(y,\hat{y}) = \frac{1}{2}[\int y^2 p(y|X=x)dy - 2\hat{y}\int yp(y|X=x)dy + \hat{y}^2]$$

$$\frac{dR}{d\hat{y}} = \frac{1}{2}[-2\int yp(y|X=x)dy + 2\hat{y}] = 0$$

$$\frac{dR}{d\hat{y}} = -\int yp(y|X=x)dy + \hat{y} = 0$$

$$\hat{y} = \int yp(y|X=x)dy$$

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

4. Feedback

1. Time Spent

I worked on this homework every day for 5 days, averaging 4 hours per day, for a total of 20 hours. This homework was far, far too long. It should have been a two-week assignment. The only reason I had time to complete it was because my other class was canceled last week due to the blizzard. If I had to do my homework for that class, I would not have been able to complete this assignment.

2. What was challenging?

It was all hard, but the programming was the most difficult because, other than the grad_checker, we had no unit tests that could validate our functions. This is especially important in a dynamically-typed language like Python. Well over half of my time debugging my program. I am a professional software engineer with 5 years of experience, so I assure you this was not due to a lack of facility with the language. Please provide unit tests so we can focus on machine learning theory and not the minutiae of our programs.

3. Other feedback

If unit tests and an extra week of time had been provided, this would have been a fun and informative assignment.

In [75]:

In [ ]: