Machine Learning and Computational Statistics, Spring 2015 Homework 1: Ridge Regression and SGD

Qingxian Lai(ql516)

1 Linear Regression

1.1 Feature Normalization

When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization, features with larger values can have a much greater effect on the final output for the same regularization cost – in effect, features with larger values become more important once we start regularizing. One common approach to feature normalization is to linearly transform (i.e. shift and rescale) each feature so that all feature values are in $[0, 1]$. Each feature gets its own transformation. We then apply the same transformations to each feature on the test set. It's important that the transformation is "learned" on the training set, and then applied to the test set. It is possible that some transformed test set values will lie outside the $[0, 1]$ interval.

Modify function feature_normalization to normalize all the features to $[0, 1]$. (Can you use numpy's "broadcasting" here?)

```
def feature_normalization(train, test):

    lmax = train.max(axis=0)
    lmin = train.min(axis=0)
    lrange = lmax - lmin

    train_normalized = (train-lmin)/lrange # use broadcasting here
    test_normalized = (test-lmin)/lrange
    return train_normalized, test_normalized
```

1.2 Gradient Descent Setup

1. Let $X \in \mathbf{R}^{m\times d+1}$ be "design matrix", where the i'th row of X is x_i. Let $y = (y_1, \ldots, y_m)^T \in \mathbf{R}^{m\times 1}$ be a the "response". Write the objective function $J(\theta)$ as an matrix/vector expression, without using an explicit summation sign.

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

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

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

3. In our search for a θ that minimizes J, suppose we take a step from θ to $\theta + \eta\Delta$, where $\Delta \in \mathbf{R}^{d+1}$ is a unit vector giving the direction of the step, and $\eta \in \mathbf{R}$ is the length of the step. Use the gradient to write down an approximate expression for $J(\theta + \eta\Delta) - J(\theta)$.

$$J(\theta + \eta\Delta) - J(\theta) \approx \frac{\eta}{m}(X\theta - y)^T X\Delta$$

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

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

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

```python
def compute_square_loss(X, y, theta):
    loss = 0    #initialize the square_loss
    m = X.shape[0]
    theta = theta.reshape(theta.shape[0], 1)

    temp = X.dot(theta)-y.reshape(m, 1)
    loss = (temp.T.dot(temp))/float(2*m)
    return loss
```

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

```python
def compute_square_loss_gradient(X, y, theta):
    m = X.shape[0]
    theta = theta.reshape(X.shape[1],1)
    grad = ((X.dot(theta)-y.reshape(m, 1)).T.dot(X))/float(m)
    return grad
```

1.3 Gradient Checker

For many optimization problems, coding up the gradient correctly can be tricky. Luckily, there is a nice way to numerically check the gradient calculation. If $J : \mathbf{R}^d \to \mathbf{R}$ is differentiable, then for any direction vector $\Delta \in \mathbf{R}^d$, the directional derivative of J at θ in the direction Δ is given by:

$$\lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon\Delta) - J(\theta - \varepsilon\Delta)}{2\epsilon}$$

We can approximate this derivative by choosing a small value of $\varepsilon > 0$ and evaluating the quotient above. We can get an approximation to the gradient by approximating the directional derivatives in each coordinate direction and putting them together into a vector. See `http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization` for details.

1. Complete the function `grad_checker` according to the documentation given.

```python
def grad_checker(X, y, theta, epsilon=0.01, tolerance=0.1):
   true_gradient = compute_square_loss_gradient(X, y, theta)
   num_features = theta.shape[0]
   e = np.identity(num_features)
   approx_grad = np.zeros(num_features)
   for i in range(num_features):
      test_theta_plus = theta+e[i]*epsilon
      test_theta_minus = theta-e[i]*epsilon
      approx_grad[i] = (compute_square_loss(X, y,
          test_theta_plus)-compute_square_loss(X, y,
          test_theta_minus))/(2*epsilon)
   approx_grad = np.array(approx_grad).reshape(true_gradient.shape)
   E_distance =
       np.sqrt((approx_grad-true_gradient).dot((approx_grad-true_gradient).T))
   return E_distance <= tolerance
```

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

```python
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]

   e = np.identity(num_features)
   approx_grad = np.zeros(num_features) #Initialize the gradient we
       approximate

   for i in range(num_features):
      test_theta_plus = theta+e[i]*epsilon
      test_theta_minus = theta-e[i]*epsilon
      approx_grad[i] = (objective_func(X, y,
          test_theta_plus)-objective_func(X, y,
          test_theta_minus))/(2*epsilon)
   approx_grad = np.array(approx_grad).reshape(true_gradient.shape)
   E_distance =
       np.sqrt((approx_grad-true_gradient).dot((approx_grad-true_gradient).T))
   return E_distance <= tolerance
```

1.4 Batch Gradient Descent

At the end of the skeleton code, the data is loaded, split into a training and test set, and normalized. We'll now finish the job of running regression on the training set. Later on we'll plot the results together with SGD results.

1. Complete `batch_gradient_descent`.

```
def batch_grad_descent(X, y, alpha=0.1, num_iter=1000,
    grad_checkerr=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
      grad_checker - 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_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
   theta_hist[0] = theta
   loss_hist[0] = compute_square_loss(X, y, theta)
   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])
      if grad_checkerr:
         if not grad_checker(X, y, theta_hist[i]):
            raise Exception("do not pass the gradient test on the
                iteration %s" % (i))
#           print "grad check failed at iteration %s" % (i)
      loss_hist[i] = compute_square_loss(X, y, theta_hist[i])
   return theta_hist, loss_hist
```

2. Starting with a step-size of 0.1 (not a bad one to start with), try various different fixed step

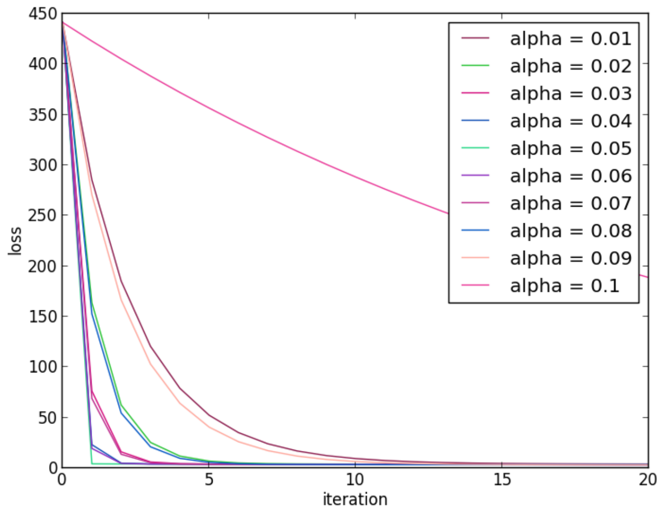

Figure 1: batch gradient descent with different step size

sizes to see which converges most quickly. Plot the value of the objective function as a function of the number of steps. Briefly summarize your findings.

As is shown in Figure 1, the loss function converges most quickly when $alpha = 0.05$

3. (Optional) Implement backtracking line search (google it), and never have to worry choosing your step size again. How does it compare to the best fixed step-size you found in terms of number of steps? In terms of time? How does the extra time to run backtracking line search at each step compare to the time it takes to compute the gradient? (You can also compare the operation counts.)

```
def batch_grad_descent_backtracking_line(X, y, alpha_init, num_iter=1000,
    grad_checkerr=False):
   """
   implement Backtracking-Armijo line search to improve the batch gradient
       desent
   """
   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
   theta_hist[0] = theta
   loss_hist[0] = compute_square_loss(X, y, theta)
   beta = 1e-4
   c = 0.5
   for i in range(1, num_iter+1):
      alpha = alpha_init
```

```
        last_loss = compute_square_loss(X,y,theta_hist[i-1])
        last_direction = compute_square_loss_gradient(X, y, theta_hist[i-1])
        theta_hist[i] = theta_hist[i-1]-alpha*last_direction

        while compute_square_loss(X,y,theta_hist[i]) > last_loss +
            alpha*beta*last_direction.dot(last_direction.T):
            alpha = alpha * c
            theta_hist[i] =
                theta_hist[i-1]-alpha*compute_square_loss_gradient(X,y,theta_hist[i-1])


        loss_hist[i] = compute_square_loss(X, y, theta_hist[i])

    return theta_hist, loss_hist
```

1.5 Ridge Regression (i.e. Linear Regression with L_2 regularization)

When we have large number of features compared to instances, regularization can help control overfitting. Ridge regression is linear regression with L_2 regularization. The objective function is

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

where λ is the regularization parameter which controls the degree of regularization. Note that the bias term is being regularized as well. We will address that below.

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

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

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

2. Implement compute_regularized_square_loss_gradient.

```
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

    """
    m = X.shape[0] # num_instances
    y = y.reshape((m, 1))
    theta = theta.reshape(X.shape[1], 1)

    grad = (1/float(m))*(X.dot(theta)-y).T.dot(X)+2*lambda_reg*theta.T
    return grad
```

3. Implement `regularized_grad_descent`.

```
def regularized_grad_descent(X, y, alpha=0.1, lambda_reg=1, num_iter=1000):
   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
   theta_hist[0] = theta
   loss_hist[0] = compute_regularized_square_loss(X, y,
       theta,lambda_reg)[0,0]

   for i in range(1,num_iter+1) :
      theta_hist[i]=theta_hist[i-1]-alpha*compute_regularized_square_loss_gradient(X,
          y, theta_hist[i-1], lambda_reg)
      loss_hist[i] = compute_square_loss(X, y, theta_hist[i])[0,0]
   return theta_hist, loss_hist
```

4. For regression problems, we may prefer to leave the bias term unregularized. One approach is to rewrite $J(\theta)$ and re-compute $\nabla_\theta J(\theta)$ in a way that separates out the bias from the other parameter. Another approach that can achieve approximately the same thing is to use a very large number B, rather than 1, for the extra bias dimension. Explain why making B large decreases the effective regularization on the bias term, and how we can make that regularization as weak as we like (though not zero).

   Answer: The bias term is actually equal to $B * b$, in which B is the constant bias dimension in the dataset and b is the last bias parameter in the θ. The regularization on the bias term is determined by λb^2. So when we set a very large B, b will become relatively small(because the bias term cannot be very large), and the λ is set to prevent the θ becoming too large, so it has little influence on the very small value b. As a result, we decrease the regularization on the bias term.

5. Start with $B = 1$. Choosing a reasonable step-size, find the θ^*_λ that minimizes $J(\theta)$ for a range of λ and plot both the training loss and the validation loss as a function of λ. (Note that this is just the square loss, not including the regularization term.) You should initially try λ over several orders of magnitude to find an appropriate range (e.g .$\lambda \in \{10^{-2}, 10^{-1}, 1, 10, 100\}$. You may want to have $\log(\lambda)$ on the x-axis rather than λ. Once you have found the interesting range for λ, repeat the fits with different values for B, and plot the results on the same graph. For this dataset, does regularizing the bias help, hurt, or make no significant difference?

   Answer: As is shown in Figure 2, the validation loss get the minimum when log(lambda) equals to -2. So we use $\lambda = 1e-2$ in the next step: searching for the right B.
   Change B between 1 5 while setting $\lambda = 1e-2, \alpha = 0.05$, we can get the optimal training loss and validation loss from the ridge gradient descent method. Then plot the result in the Figure 3:
   The Figure 3 shows: when changing the bias B, the validation loss only has a slightly increase, and the training loss remains the same. So I draw the conclusion that regularizing the bias make no significant difference.

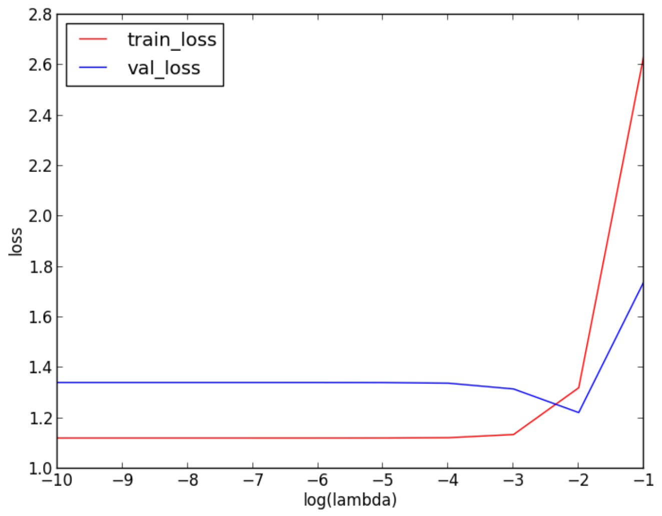

Figure 2: Redge gradient descent train loss and validation loss

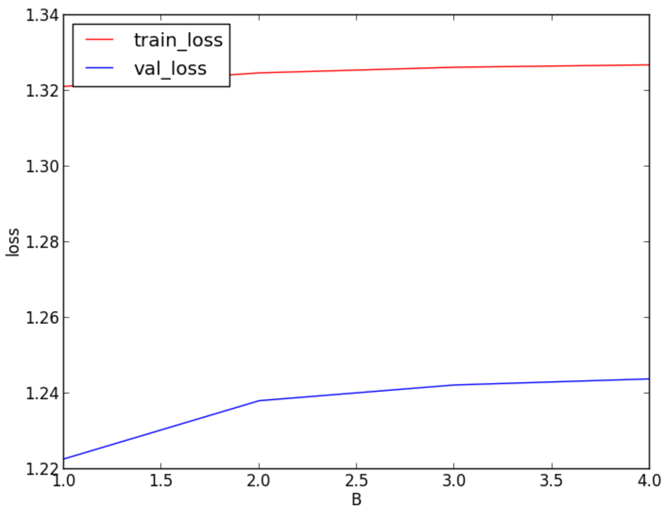

Figure 3: Train loss and validation loss when changing bias B

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

   Answer: I use the python built in package "time" to estimate the time period of the whole gradient descent process, and then divided it by iteration number (1000) to get the run time per iteration.The result is 1.6×10^{-5}second per iteration.

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

   Answers: We should use the θ the gradient descent produced, because it is the optimal. Since the Figure 3 shows the change of B has little influence on the result, I set $B = 1$. The Figure 2 shows that the validation loss get minimum when $\lambda = 0.01$, so we use it as the default value in the following functions.

1.6 Stochastic Gradient Descent

When the training data set is very large, evaluating the gradient of the loss function can take a long time, since it requires looking at each training example to take a single gradient step. In this case, stochastic gradient descent (SGD) can be very effective. In SGD, the gradient of the risk is approximated by a gradient at a single example. The approximation is poor, but it is unbiased. The algorithm sweeps through the whole training set one by one, and performs an update for each training example individually. One pass through the data is called an *epoch*. Note that each epoch of SGD touches as much data as a single step of batch gradient descent. It is also important that as we cycle through the training examples, they are in a random order, though it doesn't seem that one needs to use a different shuffling for each epoch. (You should want to investigate to confirm or refute that last claim.)

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

$$\begin{aligned}\theta &= \theta - \eta \nabla J_i(\theta) \\ &= \theta - \frac{\eta}{m}((X_i\theta - y_i)X_i + 2\lambda\theta^T)^T \\ &= \theta - \frac{\eta}{m}(X_i^T(X_i\theta - y_i)^T + 2\lambda\theta)\end{aligned}$$

2. Implement `stochastic_grad_descent`.

```
def stochastic_grad_descent(X, y, alpha=0.05, lambda_reg=0.01,
    num_iter=1000):

   num_instances, num_features = X.shape[0], X.shape[1]
   theta = np.zeros(num_features) #Initialize theta
   rand = np.arange(num_instances)
   theta_hist = []
   loss_hist = []
   theta = np.ones(num_features)
   np.random.shuffle(rand)
   for i in range(num_iter):
      theta_list = []
      loss_list = []
```

```
        if i != 0:
            alpha = 0.005/i
        else:
            alpha = 0.005

        for j in range(num_instances):
            x = X[rand[j]]
            yy = y[rand[j]]

            theta = theta - alpha * sgd_loss_gradient(x,yy,theta,lambda_reg)
            theta_list.append(theta)
            loss = sgd_loss(x,yy,theta)
            loss_list.append(loss)
        theta_hist.append(theta_list)
        loss_hist.append(loss_list)
    theta_hist = np.array(theta_hist)
    loss_hist = np.array(loss_hist)
    return theta_hist, loss_hist
```

3. Use SGD to find θ^*_λ that minimizes the ridge regression objective for the λ and B that you selected in the previous problem. Try several different fixed step sizes, as well as step sizes that decrease with the step number according to the following schedules: $\eta_t = \frac{1}{t}$ and $\eta_t = \frac{1}{\sqrt{t}}$. Plot the value of the objective function (or the log of the objective function if that is more clear) as a function of epoch (or step number) for each of the approaches to step size. How do the results compare? (Note: In this case we are investigating the convergence rate of the optimization algorithm, thus we're interested in the value of the objective function, which includes the regularization term.)

   Answer: The graph shows that the SGD algorithm takes relatively less steps(epochs) to get converge than the Ridge gradient descent no matter which step size is chosen. Compare these three types of step size, $alpha = 1/t$ and $alpha = 1/\sqrt{t}$ are more stable than the model with fixed step size.And the one with $alpha = 1/\sqrt{t}$ has a lower loss so it is the optimal step size in this scenario.

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

   Answer: I use the python built in package "time" to estimate the time period of the whole gradient descent process, and then divided it by iteration number (1000) to get the run time per iteration.The result is 2.3×10^{-3}second per iteration.

5. Comparing SGD and gradient descent, if your goal is to minimze the total number of epochs (for SGD) or steps (for batch gradient descent), which would you choose? If your goal were to minimize the total time, which would you choose?

   Answers: If our goal is to minimze the total number of epochs or steps, we shoule choose SGD algorithm because it use less epochs to converge. If our goal is to minimze the total time, we should choose the ridge gradient descent. However, I did not consider the prerformance loss of my poor implementation of SGD. It has an inner loop, which may be the reason why SGD is much slower than the RGD algorithm.

2 Risk Minimization

Recall the statistical learning framework, which $(X, Y) \sim P_{\mathcal{X}\times\mathcal{Y}}$, the **expected loss** or **"risk"** of a decision function $f : \mathcal{X} \to \mathcal{A}$ is

$$R(f) = \mathbb{E}\ell(f(X), Y),$$

and a **Bayes decision function** $f_* : \mathcal{X} \to \mathcal{A}$ is a function that achieves the *minimal risk* among all possible functions:

$$R(f_*) = \inf_f R(f).$$

Here we consider the regression setting, in which $\mathcal{A} = \mathcal{Y} = \mathbf{R}$.

1. Show that for the square loss $\ell(\hat{y}, y) = \frac{1}{2}(y - \hat{y})^2$, the Bayes decision function is a $f_*(x) = \mathbb{E}[Y \mid X = x]$. [Hint: Consider constructing $f_*(x)$, one x at a time.]

   Answer: Set $X = x$, $\widehat{y} = f(X)$,then we have:

$$\begin{aligned}
g(f(X)) &= \frac{1}{2}E[(f(X) - y)^2] \\
g(\widehat{y}|x) = g(f(X)|X = x) &= \frac{1}{2}E[(\widehat{y} - y)^2|X = x] \\
g(\widehat{y}|x) &= \frac{1}{2}E[(\widehat{y}^2 - 2\widehat{y}y + y^2)|X = x] \\
g(\widehat{y}|x) &= \frac{1}{2}(E[\widehat{y}^2|x] - 2E[\widehat{y}y|x] + E[y^2|x]) \\
g(\widehat{y}|x) &= \frac{1}{2}E[\widehat{y}^2|x] - E[\widehat{y}y|x] + \frac{1}{2}E[y^2|x] \\
g(\widehat{y}|x) &= \frac{1}{2}\widehat{y}^2 - \widehat{y}E[y|x] + \frac{1}{2}E[y^2|x] \\
\frac{dg(\widehat{y}|x)}{d\widehat{y}} &= \widehat{y} - E[y|x]
\end{aligned}$$

   Set the first order derivative to 0, we get:

$$\widehat{y} = E[y|x]$$

   which is equalvilent to:

$$f(x) = E[y|X = x]$$

2. (Optional challenge) Show that for the absolute loss $\ell(\hat{y}, y) = |y - \hat{y}|$, the Bayes decision function is a $f_*(x) = \text{median}\,[Y \mid X = x]$. [Hint: Again, consider one x at time, and you can use the following characterization of a median: m is a median of the distribution for random variable Y if $P(Y \geq m) \geq \frac{1}{2}$ and $P(Y \leq m) \geq \frac{1}{2}$.] Note: This loss function leads to "median regression", There are other loss functions that lead 'quantile regression' for any

chosen quantile.
Answer:

$$\begin{aligned} g(\widehat{y}|x) &= E[|y-\widehat{y}||x] \\ &= (y-\widehat{y})P(y>\widehat{y}|x)+(\widehat{y}-y)P(y<\widehat{y}|x) \end{aligned}$$

$$\frac{dg(\widehat{y}|x)}{d\widehat{y}} = -P(y>\widehat{y}|x)+P(y<\widehat{y}|x)$$

set the first order derivative to zero, we can get:

$$P(y>\widehat{y}|x)=P(y<\widehat{y}|x)$$

And we know that $P(y>\widehat{y}|x)+P(y<\widehat{y}|x)=1$, then:

$$P(y>\widehat{y}|x)=P(y<\widehat{y}|x)=\frac{1}{2}$$

so $\widehat{y}=median[y|x]$, which is equivalent to

$$f_*(x)=\operatorname{median}[Y \mid X=x]$$

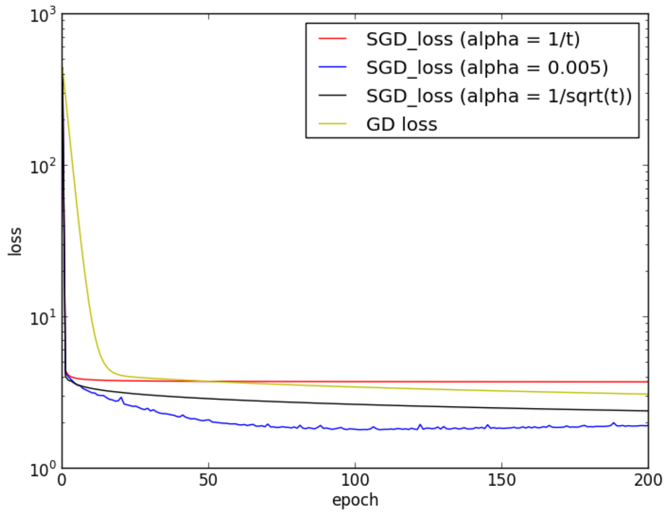

Figure 4: Stochastic Gradient Descent with different step size