Assignment 1

Benjamin Jakubowski

February 5, 2016

2.2 Gradient Descent Setup

1. Objective function

First, let $X \in \mathbb{R}^{m \times d+1}$ be the "design matrix", where the i'th row of X is x_i. Let $y = (y_1, \ldots, y_m)^T \in \mathbb{R}^{m \times 1}$ be the "response". Then the objective function $J(\theta)$ is

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

2. Gradient of J

The gradient of J is

$$\begin{aligned} J(\theta) &= \frac{1}{2m}||X \cdot \theta - y||_2^2 \\ &= \frac{1}{2m}(X \cdot \theta - y)^T (X \cdot \theta - y) \\ \implies \qquad \nabla_\theta J(\theta) &= \frac{1}{2m} 2(X \cdot \theta - y)^T \cdot X \\ &= \frac{1}{m}(X \cdot \theta - y)^T \cdot X \end{aligned}$$

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

We are interested in finding a first-order approximation for $J(\theta + \eta\Delta) - J(\theta)$. First note (from the taylor expansion) the first-order approximation of $J(\theta_2)$ around θ_1 is

$$J(\theta_2) \approx J(\theta_1) + \nabla_\theta J(\theta)^T (\theta_2 - \theta_1)$$

Thus, with $\theta_2 = \theta + \eta\Delta$ and $\theta_1 = \theta$, we have

$$\begin{aligned} J(\theta + \eta\Delta) - J(\theta) &= J(\theta) + \nabla_\theta J(\theta)^T (\theta + \eta\Delta - \theta) \\ &= J(\theta) + \nabla_\theta J(\theta)^T (\eta\Delta) \\ &= J(\theta) + \eta \nabla_\theta J(\theta)^T \cdot \Delta \end{aligned}$$

As an intuitive explanation for this result, note $\eta \nabla_\theta J(\theta)^T \cdot \Delta$ is just the directional derivative in the direction of Δ, scaled by the step size.

4. Expression for updating θ in the gradient descent algorithm

To update θ in the gradient descent algorithm, we simply use:

$$\theta \leftarrow \theta - \eta \frac{\nabla_\theta J(\theta)}{||\nabla_\theta J(\theta)||_2}$$

where

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

5. Implementing `compute_square_loss` in python

See code in appendix.

6. Verifying `compute_square_loss`

Let

$$X = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \text{ and } y = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

Then, for

$$\theta = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

$$J(\theta) = \frac{1}{2 \cdot 2} \begin{bmatrix} 0 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \frac{1}{4} \cdot 4 = 1$$

For

$$\theta = \begin{bmatrix} 0 \\ 1.5 \end{bmatrix}$$

$$J(\theta) = \frac{1}{2 \cdot 2} \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} = \frac{1}{4} \cdot 0.5 = 0.125$$

These results are confirmed using `compute_square_loss`.

7. Implementing `compute_square_loss_gradient` in python

See code in appendix.

8. Verifying `compute_square_loss`

Again, with X and y as defined in 6, for

$$\theta = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \frac{1}{2}\left(\begin{bmatrix}0 & 1\\ 2 & 3\end{bmatrix}\begin{bmatrix}0\\ 2\end{bmatrix} - \begin{bmatrix}2\\ 4\end{bmatrix}\right)^T \begin{bmatrix}0 & 1\\ 2 & 3\end{bmatrix}\\
&= \frac{1}{2}\left(\begin{bmatrix}2\\ 6\end{bmatrix} - \begin{bmatrix}2\\ 4\end{bmatrix}\right)^T \begin{bmatrix}0 & 1\\ 2 & 3\end{bmatrix}\\
&= \frac{1}{2}\begin{bmatrix}0 & 2\end{bmatrix}\begin{bmatrix}0 & 1\\ 2 & 3\end{bmatrix}\\
&= \frac{1}{2}\begin{bmatrix}4 & 6\end{bmatrix}\\
&= \begin{bmatrix}2 & 3\end{bmatrix}
\end{aligned}
$$

These results are confirmed using `compute_square_loss_gradient`.

2.3 Gradient Checker

1. Complete `grad_checker`

See code in appendix.

2. Complete generic version of `grad_checker`

See code in appendix.

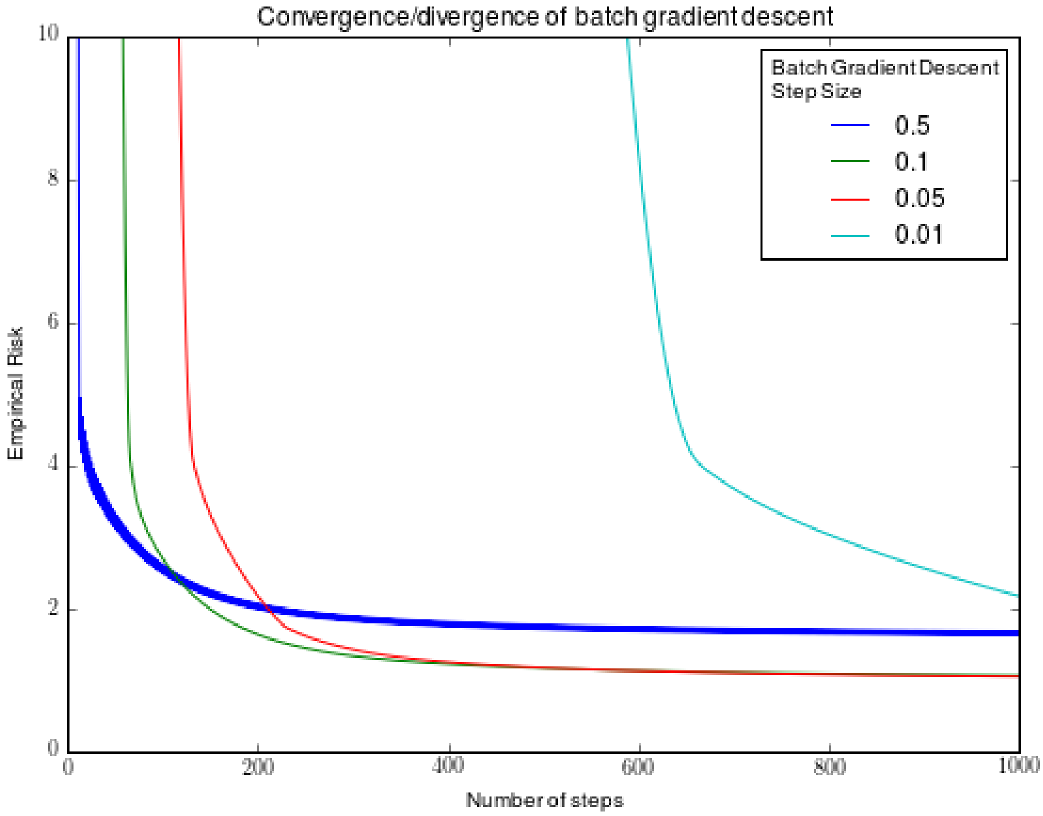
2.4 Batch Gradient Descent

1. Complete `batch_gradient_descent`

See code in appendix.

2. Experimenting with batch gradient descent stepsize

After implementing `batch_gradient_descent`, the following step sizes were tested: 1, 0.5, 0.1, 0.5, and 0.1. Plots showing the value of the objective function as a function of the number of steps for each step size are shown below:

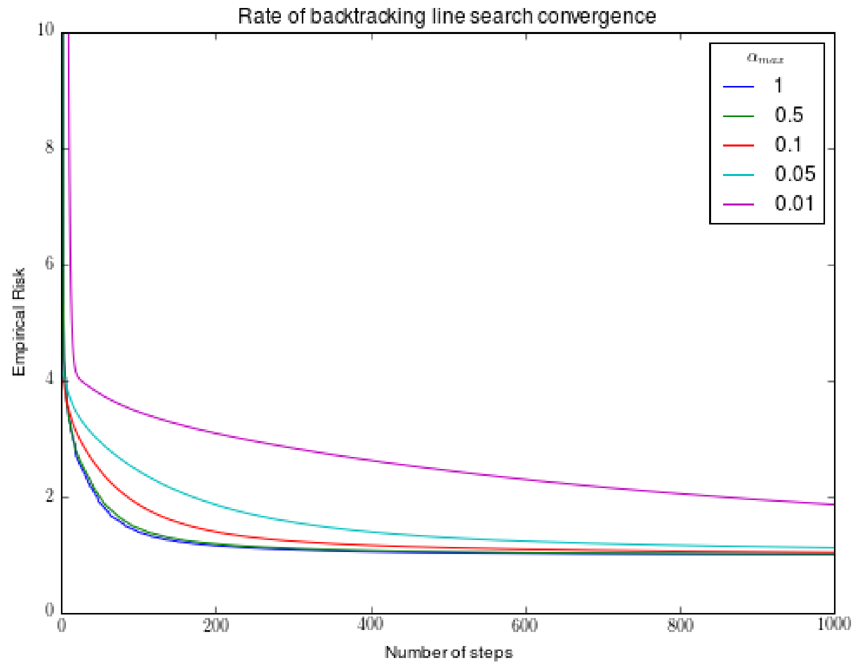


This graph indicates that for this data set:

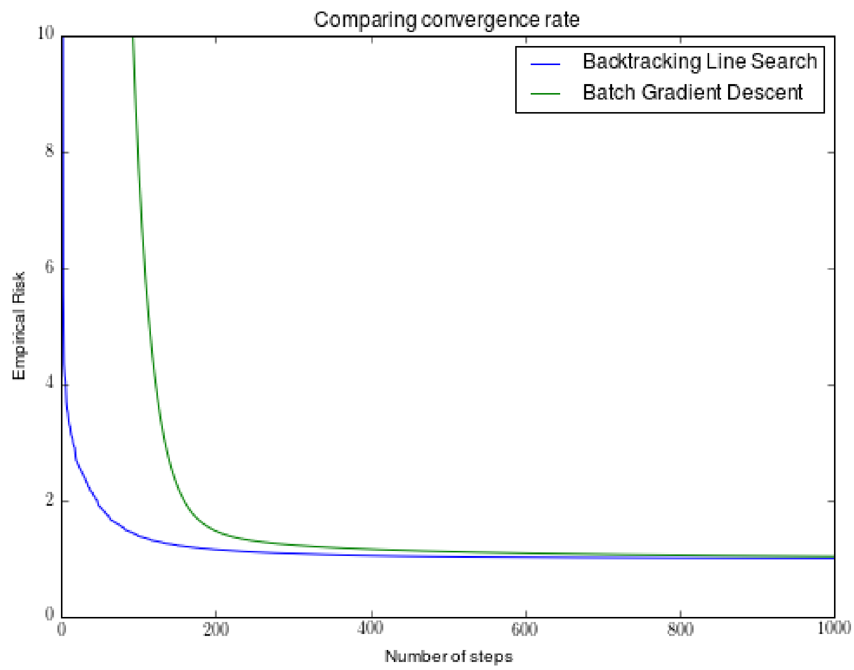
- Batch gradient descent does not converge for $\eta = 0.5$ (note the "thickness" of the blue line indicates oscillation between greater and lower empirical risk values, i.e. across the objective minimum). Additionally $\eta = 0.001$ is too small a step size to achieve convergence in 1000 iterations.
- Of the tested step sizes, $\eta = 0.1$ converged most quickly, converging to a risk-minimizing θ within approximately 300 steps.

3. Backtracking line search

Backtracking line search was implemented in batch gradient descent. Various values for the maximum step size α_{max} were tested, and the search control parameters were both set to 0.5 (per the original Armijo, 1966 paper). The Plots showing the value of the objective function as a function of the number of steps for each α_{max} are shown below:



Compared to batch gradient descent with the best fixed step size ($\alpha = 0.1$), it is apparent that backtracking line search converges more quickly:



It is apparent that backtracking line search converges in approximately half the number of steps. However, this improved performance comes at a time cost. Using `%timeit` magic in iPython on my machine, the following runtimes were obtained:

Method	Runtime
Batch gradient descent ($\alpha = 0.1$)	`10 loops, best of 3: 20.2 ms per loop`
Backtracking line search	`10 loops, best of 3: 112 ms per loop`

Thus, given these results, it appears to take approximately 5 times longer to run backtracking line search over batch gradient descent with a fixed step size. Given, however, that the fixed step size was determined empirically by testing multiple different fixed step sizes, backtracking line search produces an improvement in performance in time.

2.5 Ridge Regression

1. Gradient of $J(\theta)$

First, note the ridge regression objective function expressed in matrix/vector notation is

$$\begin{aligned} J(\theta) &= \frac{1}{2m}||X\theta - y||_2^2 + \lambda||\theta||_2^2 \\ &= \frac{1}{2m}(X\theta - y)^T(X\theta - y) + \lambda\theta^T\theta \end{aligned}$$

Thus, the gradient of $J(\theta)$ is

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

Using this gradient, the update rule becomes

$$\theta \leftarrow \theta - \eta\nabla_\theta J(\theta) = \theta - \eta\left[\frac{1}{m}(X\theta - y)^T X + 2\lambda\theta\right]$$

2. Implementing `COMPUTE_REGULARIZED_SQUARE_LOSS_GRADIENT`

See code in appendix.

3. Implementing `REGULARIZED_GRAD_DESCENT`

See code in appendix.

4. Decreasing effective regularization of bias term

To decrease the effective regularization of the bias term, we can increase B. To see this, consider the design matrix $X \in \mathbb{R}^{m \times n+1}$. We can rewrite X as

$$X = [D\ B]$$

where D is the data, and B is the bias vector.
Then, the objective function becomes

$$\begin{aligned} J(\theta) &= \frac{1}{2m}||X\theta - y||_2^2 + \lambda||\theta||_2^2 \\ &= \frac{1}{2m}||\,[D\ B]\,\theta - y||_2^2 + \lambda||\theta||_2^2 \end{aligned}$$

Now let B_0 be given, and let θ_0 be the solution to the ridge regression problem with $X = [D\ B_0]$.

Then, let $B_1 = cB_0$ for some $c >> 1$, and let θ_1 be the ridge regression solution with $X = [D\ B_1]$. Then it is apparent $\theta_1[n+1] << \theta_0[n+1]$. Thus, by increasing c, we can correspondingly decrease the bias coefficient, and in doing see decrease the effective regularization on the bias term such that $\theta[n+1]^2 < \epsilon$ for any $0 < \epsilon$.

5. Finding θ_λ^* that minimizes $J(\theta)$

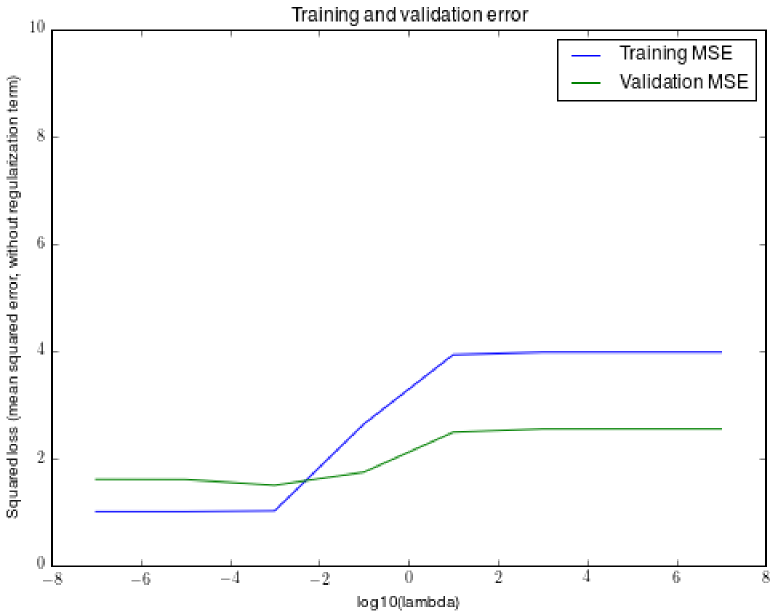
The following approach was taken to find θ_λ^* that minimizes $J(\theta)$:

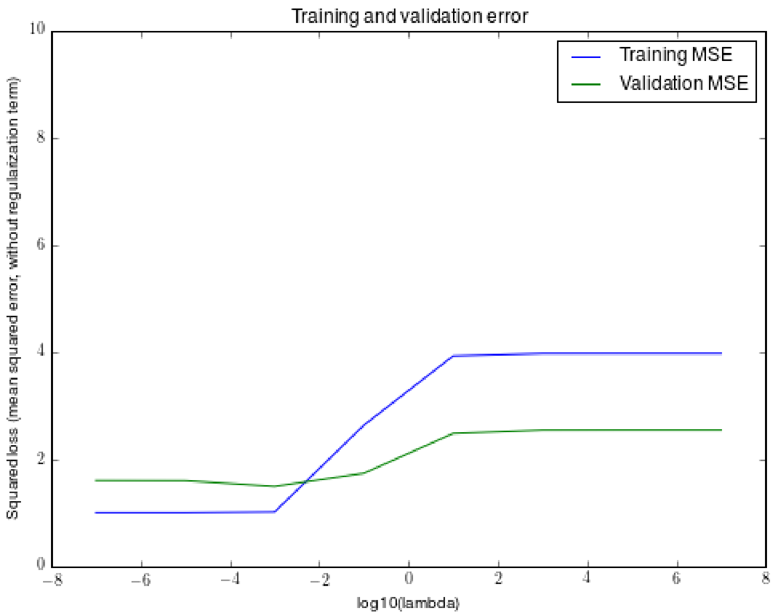
1. For candidate λ's, batch gradient descent with backtracking line search was used to find the θ_λ that minimized the ridge regression loss function.

2. Then, using this θ_λ, $\frac{1}{2}MSE$ was calculated for both the training and validation set.

3. Steps 1 and 2 were iterated for increasingly fine-grained λ candidate lists, until the optimal λ (in the sense of minimum validation loss) was obtained to three decimal places. This yielded θ_λ^*.

The two plots below show the first two interations:

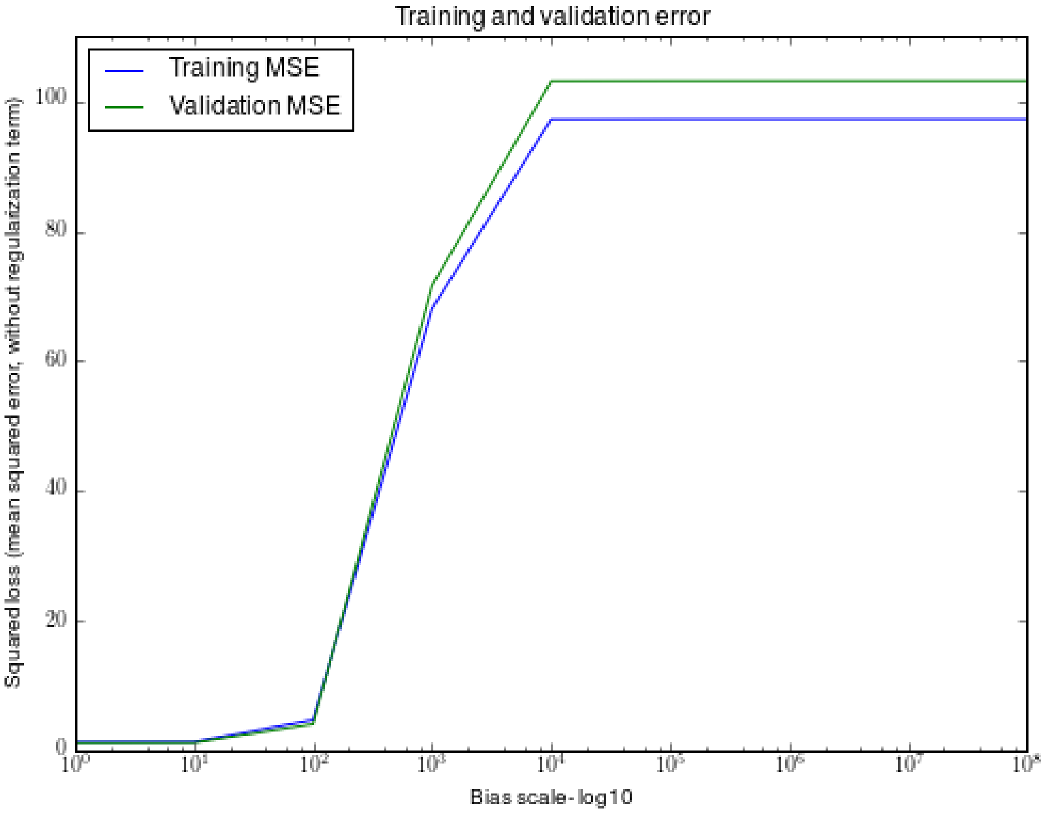




Ultimately, the optimal value was $\lambda = .0122$.

6. Comparing different values of B

Next, setting $\lambda = 0122$, ridge regression with fit with different values of B. Plots of $\frac{1}{2}MSE$ on the training and validation set are shown below:



From the plots, it is apparent that regularizing the bias helps the model- the best model has $B = 1$, and as B increases (i.e. the regularization on the bias term decreases) the model performance decreases.

7. Average time it takes to compute a single gradient step

Again, the following runtime for batch gradient descent on ℓ_2-regularized regression was obtained using `%timeit` magic in iPython on my machine:

Method	Runtime
Batch gradient descent for ridge regression	`10 loops, best of 3:  30.5 ms per loop`

Since each loop ran with 1000 iterations (steps), each step is computed in approximately 3 μs.

8. Optimal θ for deployment

I would choose the θ fit with $\lambda = 0.012$ and $B = \mathbf{1}$. This is because these values minimized the validation set empirical risk. The value of θ was found, and the first three terms and last term are

$$\begin{bmatrix} -1.13963079 & 0.48238988 & 1.27016416 & \ldots & -1.5816435 \end{bmatrix}$$

The remaining coefficients varied between 2.287 and -3.607.

2.5 Stochastic Gradient Descent (SGD)

1. Update rule for θ in SGD

In SGD, the gradient of the risk is approximated by the gradient at a single example. Thus, this gradient (for ridge regression) is

$$\begin{aligned} \nabla_\theta J_{SGD}(\theta) &= \nabla_\theta \left[\frac{1}{2}(x_i^T \cdot \theta - y_i)^2 + \lambda \theta^T \theta \right] \\ &= (x_i^T \cdot \theta - y) \cdot x_i + 2\lambda\theta \end{aligned}$$

Thus, the update rule is

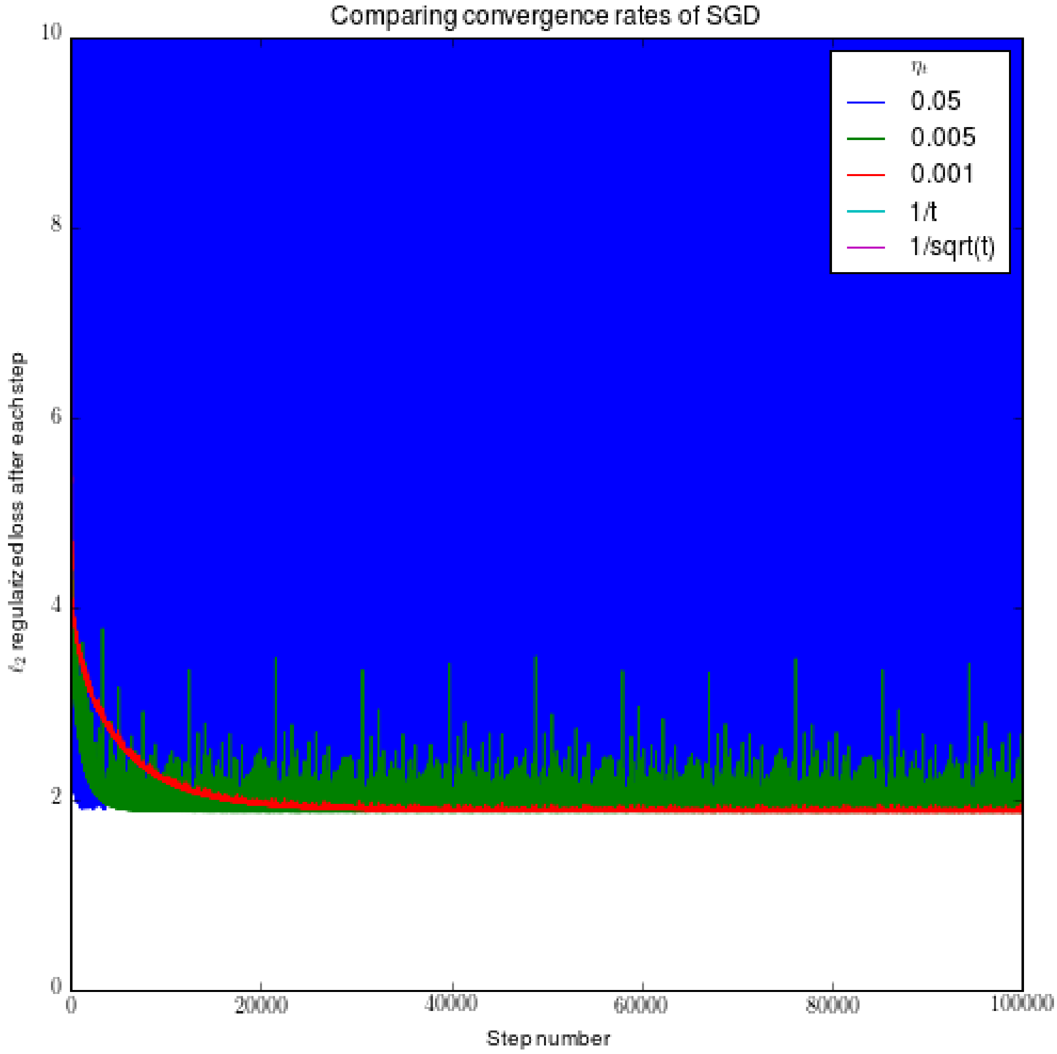
$$\theta \leftarrow \theta - \eta \left[(x_i^T \cdot \theta - y) \cdot x_i + 2\lambda\theta \right]$$

2. Implementing `stochastic_grad_descent`

See code in appendix.

3. Using SGD to find θ^*_λ

In this section, we compare the convergence of SGD to an optimal θ^*_λ using fixed step sizes and step sizes that decrease with the step number. Specifically, fixed step sizes $\eta_t = 0.05$ and 0.005 were tested, along with the following step sizes as functions of t: $\eta_t(t) = \frac{1}{t}$ and $\eta_t(t) = \frac{1}{\sqrt{t}}$. Plots showing the ℓ_2 regularized square loss for each of these step sizes are shown below:



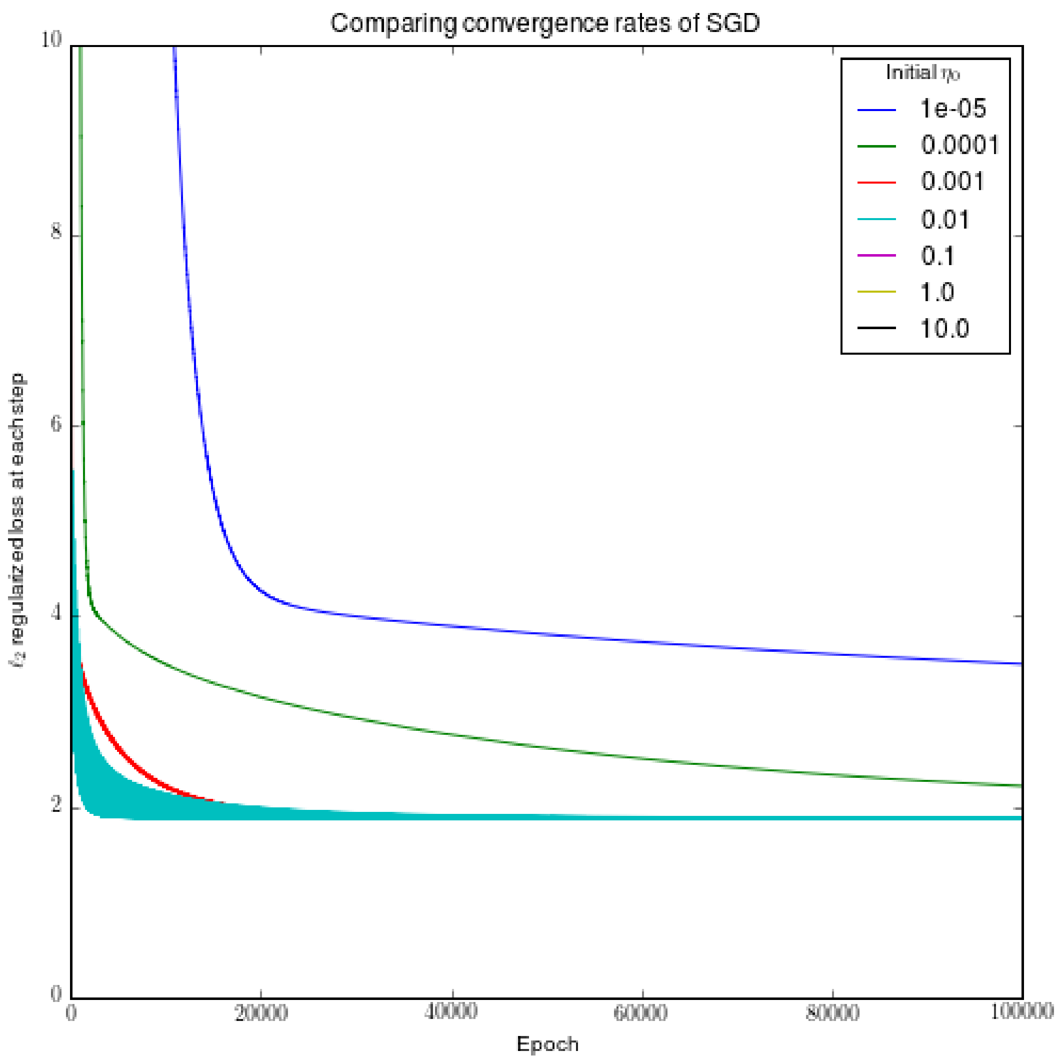
To compare the results, note:

- $\eta_t = 0.05$ does not converge- this fixed step size is too large, and as a results we get wild oscillations (resulting in the plot appearing as a solid block).
- $\eta_t = 0.005$ is somewhat noisy- there is an apparent cycle, with updates of θ over-fitting to the single instance x_i, moving θ away from empirical risk minimizer.
- At the resolution selected for the plot above, $\eta_t = 0.001$ appears relatively stable, though at higher resolution we'd expect to observe a similar cycle.

- Neither $\eta_t = \frac{1}{t}$ or $\eta_t = \frac{1}{\sqrt{t}}$ converge. This is likely due to the initially large step size (i.e. for small t η_t is relatively large, compared to our converging fixed steps size, which is on the order of 0.001). Note this claim is further supported by results in 2.5.4.

4. Implementing an additional stepsize function

Next, we try a step size rule of the form $\eta_t = \frac{\eta_0}{1+\eta_0 \lambda t}$. Using $\eta_0 \in [10^{-5}, 10^{-4}, ...10^{1}]$, we obtain the following convergence results:



Comparing these results, it is apparent that the convergence rate increases with η_0 up to $\eta_0 = 0.01$. However, all tested $\eta_0 \geq 0.1$ diverged.

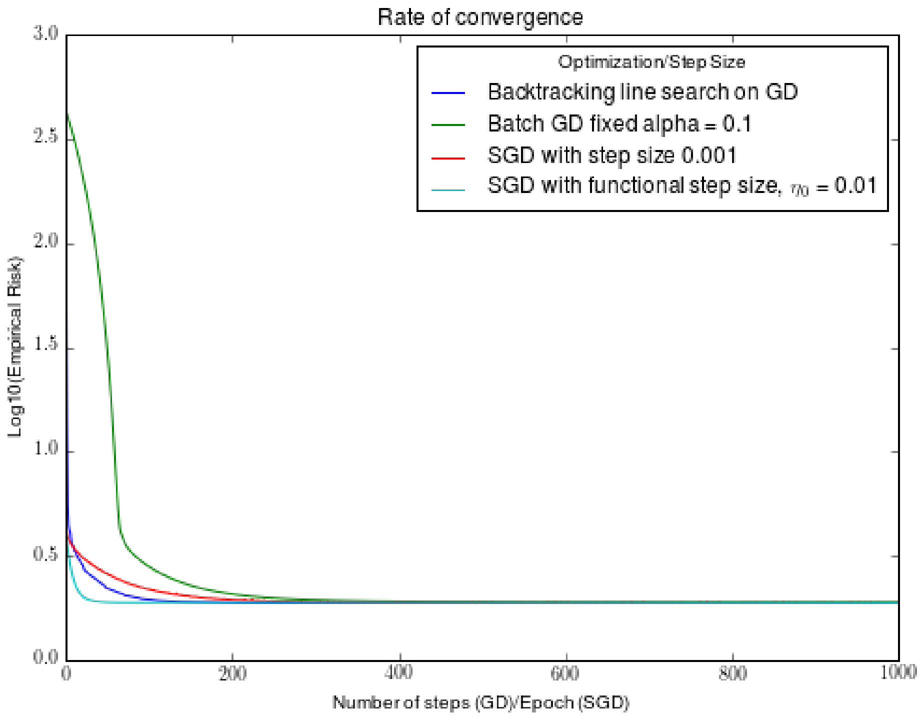
5. Comparing runtime

Again, the following runtimes for SGD on ℓ_2-regularized regression was obtained using `%timeit` magic in iPython on my machine:

Stepsize	Runtime	Avg. per epoch
0.001	`1 loops, best of 3: 2.4 s per loop`	2.4 ms
0.005	`1 loops, best of 3: 2.49 s per loop`	2.49 ms
$1/t$	`1 loops, best of 3: 2.92 s per loop`	2.92 ms
$1/\sqrt{t}$	`1 loops, best of 3: 3.07 s per loop`	3.07 ms
$\frac{\eta_0}{1+\eta_0 \lambda t}$, with $\eta_0 = 0.1$	`1 loops, best of 3: 2.68 s per loop`	2.68 ms

6. Comparing SGD and Gradient Descent

First, lets compare the convergence rate. Plots of log_{10}(Empirical Risk) (ℓ_2 regularized risk on the entire training set) are shown for several versions of SGD and Gradient Descent (all with $\lambda_{reg} = 0.0122$, as discussed previously).



From this plot, it appears optimal convergence is obtained using SGD with functional step size $\eta_t = \frac{\eta_0}{1+\eta_0 \lambda t} = \frac{0.01}{1+0.01 \lambda t}$. However, batch GD with backtracking line search achieves similar convergence.

Next, let's compare runtimes. Using `%timeit`, we observed a step in SGD was on the order of 3 μs (with backtracking line search increasing runtime by approximately 5×), while an epoch in SGD was on the order of 3 ms. This indicates an epoch of SGD is on the order of 1000 times slower than a step of SGD. This, however, is not unexpected- our implementation of SGD evaluates the loss (over the entire training set) at each step. This is anticipated to substantially increase runtime.
Regardless, as implemented, I would select the following constraints:

Constraint	Algorithm
Minimized total time	Batch gradient descent with backtracking line search
Minimized total number of steps/epochs	SGD with functional step size, $\eta_0 = 0.01$

3. Risk Minimization

1. Posterior Mean: Minimum MSE estimator

Let f be an arbitrary decision function, and let x be given. Then, conditioning on $X = x$,

$$\begin{aligned}\frac{1}{2}E\left[(Y - f(X))^2 \,|X = x\right] =& \frac{1}{2}E\left[(Y - E[Y|X] + E[Y|X] - f(X))^2 \,|X = x\right] \\ =& \frac{1}{2}E\left[(Y - E[Y|X])^2 \,|X = x\right] + \frac{1}{2}E\left[(E[Y|X] - f(X))^2 \,|X = x\right] \\ &+ E\left[(Y - E[Y|X])(E[Y|X] - f(X))|X = x\right]\end{aligned}$$

Next, noting the last term $E\left[(Y - E[Y|X])(E[Y|X] - f(X))|X = x\right] = 0$ yields:

$$\frac{1}{2}E\left[(Y - f(X))^2 \,|X = x\right] = \frac{1}{2}E\left[(Y - E[Y|X])^2 \,|X = x\right] + \frac{1}{2}E\left[(E[Y|X] - f(X))^2 \,|X = x\right]$$

Next, by the non-negativity of expectation, we have

$$\begin{aligned}\frac{1}{2}E\left[(Y - f(X))^2 \,|X = x\right] &= \frac{1}{2}E\left[(Y - E[Y|X])^2 \,|X = x\right] + \frac{1}{2}E\left[(E[Y|X] - f(X))^2 \,|X = x\right] \\ &\geq \frac{1}{2}E\left[(Y - E[Y|X])^2 \,|X = x\right]\end{aligned}$$

Finally, using iterated expectations, we have

$$\begin{aligned} E\left[\frac{1}{2}E\left[(Y - f(X))^2 \,|X\right]\right] &\geq \left[\frac{1}{2}E\left[(Y - E[Y|X])^2 \,|X = x\right]\right] \\ \implies \qquad \frac{1}{2}E\left[(Y - f(X))^2\right] &\geq \frac{1}{2}E\left[(Y - E[Y|X])^2\right]\end{aligned}$$

Hence the conditional expectation minimizes the mean squared error.