Lihan Yao ML, Spring 2017

# Homework 1: Ridge Regression and SGD

**Due: Thursday, February 2, 2017, at 10pm (Submit via GradeScope)**

## 1 Introduction

## 2 Linear Regression

### 2.1 Feature Normalization

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

```
def feature_normalization(train, test):
    for col in range(train.shape[1]):
        minimum = np.nanmin(train[:,col])
        train[:, col] -= minimum
        test[:, col] -= minimum

        max_ = np.nanmax(train[:,col])
        train[:,col] /= max_
        test[:,col] /= max_

    return train, test
```

### 2.2 Gradient Descent Setup

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

The objective function may be expressed in vector notation $J(\theta) = r^T r$ where residual $r = X\theta - y$. $\frac{1}{2}$ is multiplied for a simpler expression while computing the gradient, and $\frac{1}{m}$ is multiplied to ensure consistency for different sized sets.

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

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

Referencing Felippa's Matrix Calculus (C.2.2.), the derivative of a scalar $J(\theta)$ with respect to vector $\theta$ is a vector whose $i$th component is $\frac{\partial J(\theta)}{\partial \theta_i}$.

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

In vector notation this is:

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

3. In our search for a $\theta$ that minimizes $J$, suppose we take a step from $\theta$ 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 $J(\theta + \eta\Delta) - J(\theta)$. [This approximation is called a "linear" or "first-order" approximation.]
Let $r = X\theta - y$.

$$\begin{aligned} J(\theta + \eta\Delta) - J(\theta) &= (r + \eta X\Delta)^T(r + \eta X\Delta) - r^T r \\ &= \eta(r^T X\Delta + (X\Delta)^T r + \eta(X\Delta)^T X\Delta) \\ &= \frac{\eta}{2}\nabla_\theta J(\theta)^T\Delta + \frac{\eta}{2}\Delta^T\nabla_\theta J(\theta) + \eta^2\|X\Delta\|^2 \\ &= \eta\Delta^T\nabla_\theta J(\theta) + \eta^2\|X\Delta\|^2 \\ &\approx \eta\Delta^T\nabla_\theta J(\theta) \end{aligned}$$

Where the last equality follows for small $\eta$, and the second to last equality follows from scalar addition.

4. Write down the expression for updating $\theta$ in the gradient descent algorithm. Let $\eta$ be the step size.

Denote by $\theta_i$ the vector at $i$th step of GD.

$$\begin{aligned} \theta_{i+1} &= \theta_i - \frac{\eta}{m}\nabla_\theta J(\theta_i) \\ &= \theta_i - \frac{\eta}{m}X^T(X\theta_i - y) \end{aligned}$$

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

```
def compute_square_loss(theta, X, y):
    residual = np.dot(theta, X.T)-y
    return (np.dot(residual,residual))/(2*X.shape[0])
```

6. Modify the function `compute_square_loss_gradient`, to compute $\nabla_\theta J(\theta)$. You may again want to use a small dataset to verify that your `compute_square_loss_gradient` function returns the correct value.

```
def compute_square_loss_gradient(theta, X, y):
    pred = np.dot(X, theta)
    residual = (pred-y)
    return np.dot(residual, X)/(X.shape[0])
```

## 2.3 Gradient Checker

1. Complete the function `grad_checker` according to the documentation given. Alternatively, you may complete the function `generic_grad_checker so` that it works 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. Note: Running the gradient checker takes extra time. In practice, once you're convinced your gradient calculator is correct, you should stop calling the checker so things run faster.

```
def unit_vec(num_features):
    z = np.zeros(num_features-1)
    for i in range(num_features):
        yield np.insert(z, i, 1)

def grad_checker(theta, X, y,  epsilon=0.01, tolerance=1e-4):
    true_gradient = compute_square_loss_gradient(theta, X, y)
    num_features = theta.shape[0]

    approx_grad = []
    for u in unit_vec(num_features):
        approx_grad.append((compute_square_loss(theta+epsilon*u, X, y) -
                            compute_square_loss(theta-epsilon*u, X, y))/(2*epsilon))

    difference = np.linalg.norm(np.array(approx_grad)-true_gradient)
    correct =  difference <= tolerance
    if not correct:
        print(correct, difference)
        raise Exception('incorrect gradient')

    return correct
```

## 2.4 Batch Gradient Descent

1. Complete `batch_gradient_descent`.

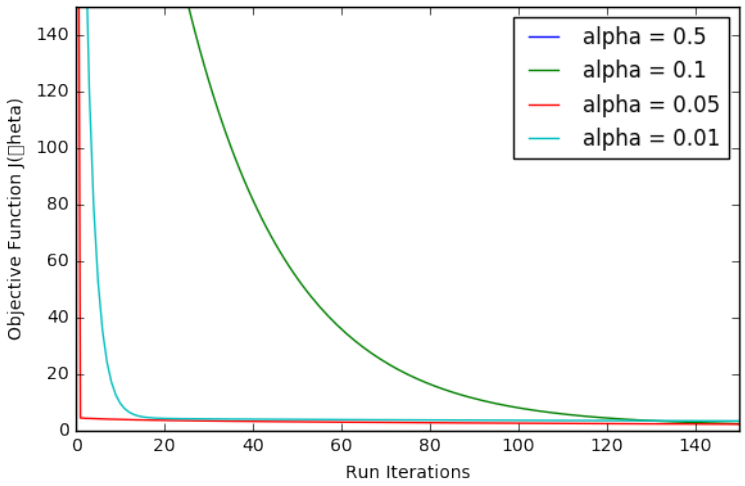
```
def batch_grad_descent(
X, y, alpha=0.1, num_iter=1000, check_gradient=False, l_search = True):

    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
    a_hist = [alpha]

    for i in range(num_iter):

        loss = compute_square_loss(theta, X, y)
        gradient = compute_square_loss_gradient(theta, X, y)

        #check terminal conditions: obj. func stop improving or gradient too small
        if i != 0 and (loss >= loss_hist[i-1] or np.dot(gradient.T, gradient) <= 1e-3):
            print(a_hist)
            return(loss_hist[0:i], theta_hist[0:i,:])

        loss_hist[i] = loss # record loss

        if check_gradient:
            grad_checker(theta, X, y)
            #gen_grad_checker(compute_square_loss_gradient, compute_square_loss)
        if l_search: #line search call
            alpha= backtrack_line_search(loss, gradient, X, y, theta, alpha)
            a_hist.append(round(alpha,4))

        theta -= alpha*gradient
        theta_hist[i,:] = theta

    return loss_hist, theta_hist
```

1. Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge[1]. Starting with a step-size of 0.1, try various different fixed step sizes to see which converges most quickly and/or which diverge. As a minimum, try step sizes 0.5, 0.1, .05, and .01. Plot the value of the objective function as a function of the number of steps for each step size. Briefly summarize your findings.

For large step sizes like 0.5, the objective function diverges. As the step size approaches 0.05, we see that convergence occurs faster and faster. For step sizes much smaller than 0.05 (I have tried the range as low as 0.001), convergence occurs slowly. Starting from the top left, these are plots of

[1]For the mathematically inclined, there is a theorem that if the objective function is convex, differentiable, and Lipschitz continuous with constant $L > 0$, then gradient descent converges for fixed step sizes smaller than $1/L$. See https://www.cs.cmu.edu/~ggordon/10725-F12/scribes/10725_Lecture5.pdf, Theorem 5.1.

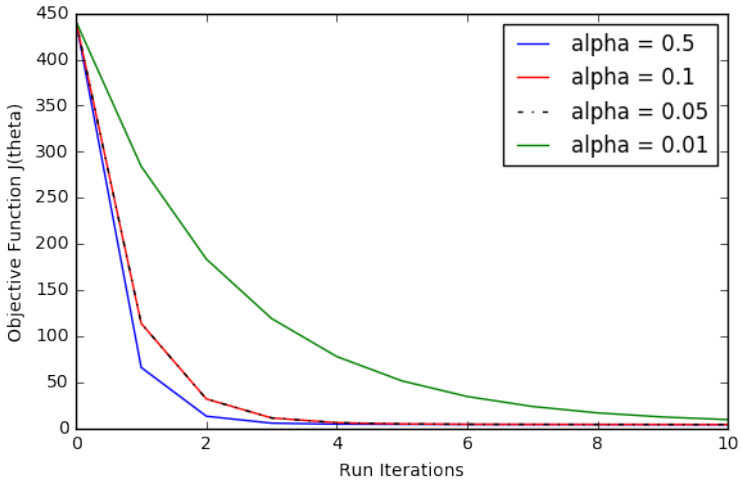
$\alpha = 0.5, 0.1, 0.05, 0.01$.




2. (Optional, but recommended)

After a few GD iterations, the backtracking line search yields $\eta$ sizes very close to $\eta = 0.05$. With this algorithm, previously diverging step sizes such as 0.5 now converges quickly, and most step sizes larger than optimal now converges almost as quickly as $\eta = 0.05$. If the initial step size is lower than optimal, nothing is changed.

With a time decorator around the ridge regression batch GD, the backtrack line search adds about 0.003-0.005 seconds extra over a GD of 1000 iterations. The initial $\eta$ optimization takes at most 0.001 seconds (this was measured by running one iteration with a slightly large initial step-size then turning off the line search).




---

```
def backtrack_line_search(fx, gradient, X, y, theta, alpha):
    tau = c = 0.5
```

```
    p = -gradient
    m = np.dot(p, gradient)
    t = c*m

    step = (compute_square_loss((theta.T + alpha*p).flatten(), X, y))

    #  Armijo-Goldstein cond: f(x) - f(x+ alpha_j * p) >= alpha_j * t
    while step- fx > alpha*t :
        alpha *= tau
        step = (compute_square_loss((theta.T + alpha*p).flatten(), X, y))

    return alpha
```

### 2.5 Ridge Regression (i.e. Linear Regression with $L_2$ regularization)

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

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

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

To update $\theta$,

$$\theta_{i+1} = \theta_i(1 - 2\eta\lambda) - \frac{\eta}{m}X^T(X\theta_i - y)$$

2. Implement `compute_regularized_square_loss_gradient`.

```
def compute_regularized_square_loss_gradient(theta, X, y, lambda_reg):
    residual = np.dot(X, theta) -y

    if len(X.shape)==1: # handles SGD calls
            return X*residual + theta*lambda_reg*2

    return (np.dot(residual, X )/X.shape[0] + 2*lambda_reg*theta)
```

3. Implement `regularized_grad_descent`.

```
def regularized_grad_descent(X, y, alpha=0.1, lambda_reg=10e-7, num_iter=1000,
                             check_gradient=False, l_search = True):

    (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) #initialize loss_hist
a_hist = [alpha]

for i in range(num_iter):

    loss = r_sq_loss(theta, X, y, lambda_reg)
    loss_hist[i] = loss

    gradient = compute_regularized_square_loss_gradient(theta, X, y, lambda_reg)

    if check_gradient:
        gen_grad_checker(compute_square_loss_gradient, compute_square_loss)
    if l_search:
        alpha= backtrack_line_search(loss, gradient, X, y, theta, alpha)
        a_hist.append(round(alpha,4))

    theta -= alpha*gradient
    theta_hist[i,:] = theta

return loss_hist, theta_hist
```

---

4. For regression problems, we may prefer to leave the bias term unregularized. One approach is to change $J(\theta)$ so that the bias is separated out from the other parameters and left unregularized. 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).

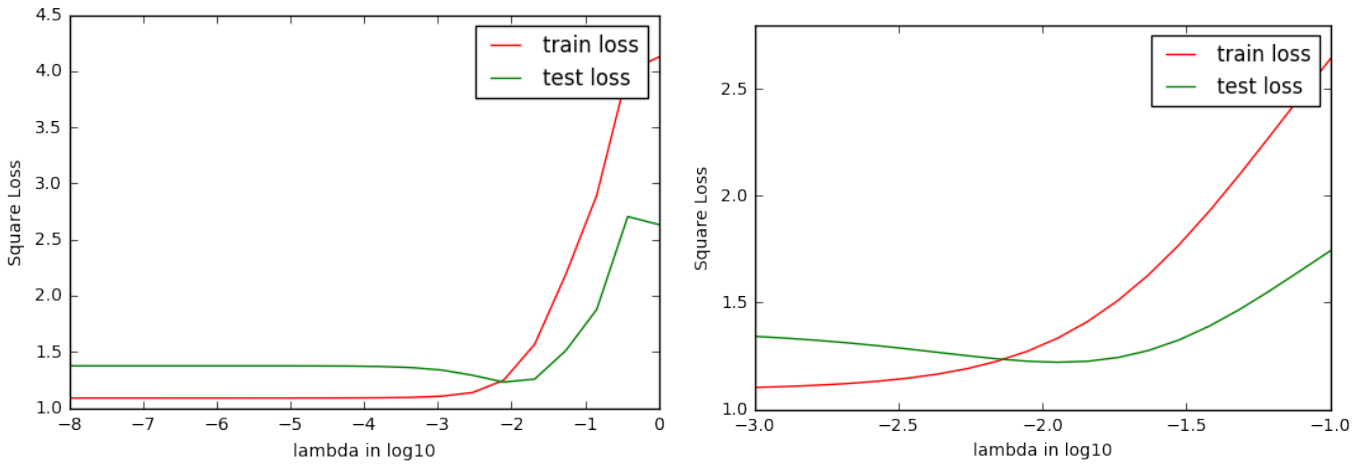
From the expression of the regularized objective function, the regularization term $\lambda\theta^T\theta$ contributes $\lambda B^2$ error from the bias component of $\theta$. For sufficiently large $B$, $B^2$ renders $\lambda$ insignificant during the computation of $J_{reg}(\theta)$. However we see that this regularization coefficient is always present for any $B$.

5. (Optional) Develop a formal statement of the claim in the previous problem, and prove the statement.

6. (Optional) Try various values of $B$ to see what performs best in test.

7. Now fix $B = 1$. Choosing a reasonable step-size (or using backtracking line search), find the $\theta^*_\lambda$ that minimizes $J(\theta)$ over a range of $\lambda$. You should plot the training loss and the test loss (just the square loss part, without the regularization, in each case) as a function of $\lambda$. Your goal is to find $\lambda$ that gives the minimum test loss. It's hard to predict what $\lambda$ that will be, so you should start your search very broadly, looking over several orders of magnitude. For example, $\lambda \in \{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\}$. Once you find a range that works better, keep zooming in. You may want to have $\log(\lambda)$ on the $x$-axis rather than $\lambda$.

From the graphs, $\lambda = 10e - 2$ gives the minimum test loss.

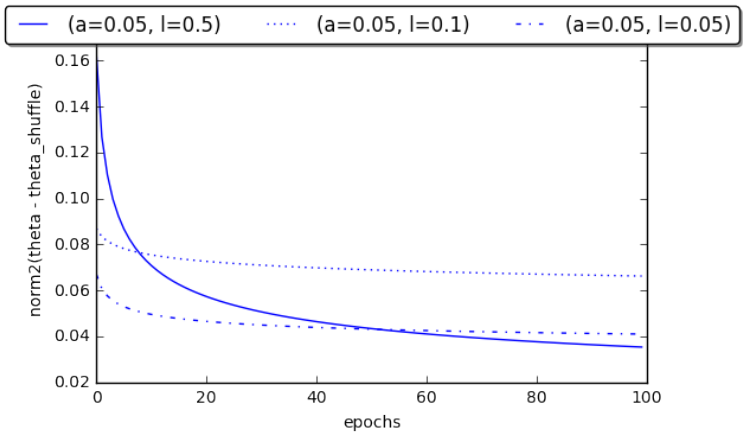



8. What $\theta$ would you select for deployment and why?

I would deploy $\theta^*_{10e-2}$ because at $\lambda = 10^{-2}$, not only does the test set incur the least loss there, but by using the test set, we know this great performance is achieved with minimal overfitting (it generalizes well).

## 2.6 Stochastic Gradient Descent

Denote by $\theta_{\text{shuffle}}$ to mean weights resulting from shuffling the samples at every epoch, and $\theta$ to mean once at the beginning of SGD. After experimenting with a range of $\eta$ and $\lambda$, at some specific epoch $i$, shuffling the sample data at epoch $i$ has minimal effects on their difference $d = ||\theta - \theta_{\text{shuffle}}||_2$. $d$ has been observed to range from 0 to 0.2, and decays as number of epochs increase. As $\lambda$ increases, this $d$ decay occurs more prominently. In the figure below, different decay rates are shown by varying $\lambda$; $\eta$ is fixed at 0.05 and $\lambda \in \{0.5, 0.1, 0.05\}$.




The controlled behavior of $d$ leads me to believe convergence (of ridge regression objective function) occurs at nearly the same rate, since their differences are negligible at later epochs.

1. *W*rite down the update rule for $\theta$ in SGD for the ridge regression objective function.

For given time step $t$ and row $i$ (let $X_i$ denote the $i$th row vector of $X$):

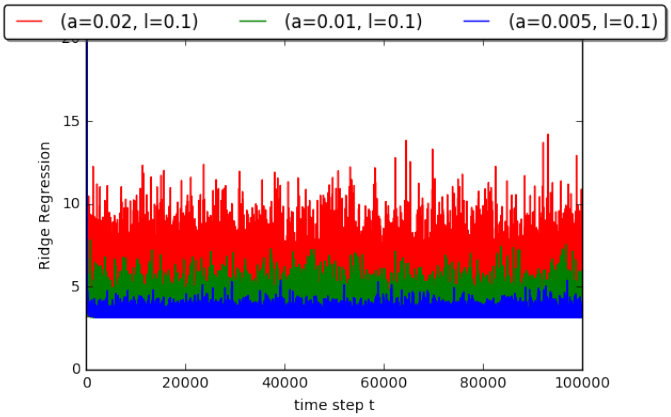
$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(X_i, \theta_t) = \theta_t - \frac{\eta}{m} X_i^T (X_i \theta_t - y_i) - 2\lambda\theta_t$$

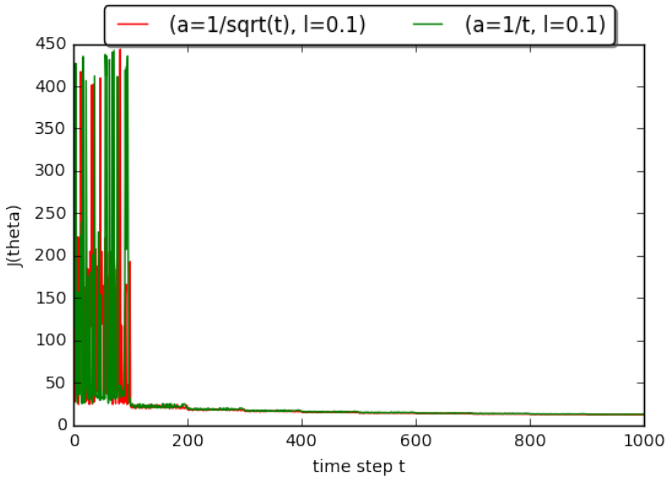
2. Implement `stochastic_grad_descent`. (Note: You could potentially reuse the code you wrote for batch gradient, though this is not necessary. If we were doing minibatch gradient descent with batch size greater than 1, you would definitely want to use the same code.)

```
def r_sq_loss(theta, X, y, lambda_reg):
    return compute_square_loss(theta, X,y) + lambda_reg*np.dot(theta.T, theta)

def stochastic_grad_descent(X, y, alpha, lambda_reg=10e-3,
                                                    num_iter=100, epi = 10e-3):
    (num_instances, num_features) = X.shape
    theta = np.ones(num_features) #Initialize theta
    theta_hist = np.zeros((num_iter, num_features))  #Initialize theta_hist
    loss_hist = np.zeros((num_iter, num_instances)) #initialize loss_hist

    s = [i for i in range(num_instances)]
    S =  np.random.permutation(s)
    for i in range(num_iter):

        for j in S:
            loss = r_sq_loss(theta, X, y, lambda_reg)
            loss_hist[i,j] = loss

            gradient = compute_regularized_square_loss_gradient(theta, X[j,:], y[j], la

            if i != 0 or j != 0:
                step = alpha/ (i*num_instances + j)

            theta -= step * gradient
            theta_hist[i] = theta
    return loss_hist, theta_hist
```

3. question omitted

The first graph displays time step $t$ vs the ridge regression objective function $J_{reg}(\theta)$ for fixed $\eta$ sizes 0.05 and 0.005. $\lambda = 0.1$, and samples are shuffled once at the beginning of SGD. Due to using a magnitude smaller step size, SGD still has not completely converged after 10 epochs with $\eta = 0.005$ The second graph is produced with the same parameters, except with step sizes $1/t$and$1/\sqrt{t}$. Updated time steps causes SGD to converge almost entirely after one epoch.



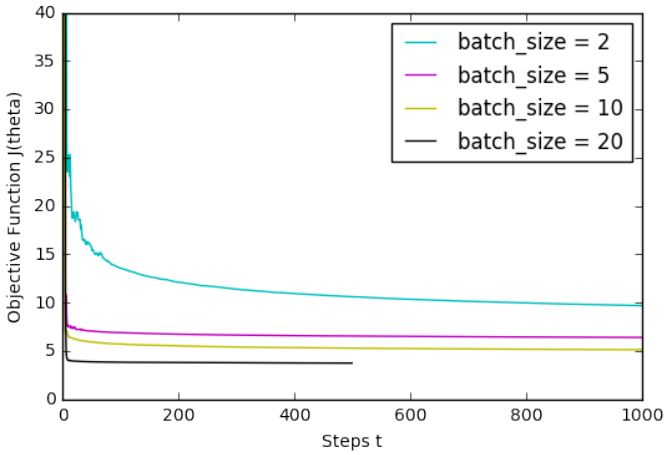

4. Estimate the amount of time it takes on your computer for a single epoch of SGD.

I have observed a small range of possible times for both setups, with differing execution times even under the same parameters. With a timeit decorator used on SGD, one epoch takes around 0.005 seconds. With the same decorator on GD (with backtrack line search), one epoch (iteration) takes around 0.009 seconds.

5. Comparing SGD and gradient descent, if your goal is to minimize 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?

For total run time, it depends on training data size. With a lot of samples, we can expect SGD to be faster than GD since it begins updating the gradient on a sample-by-sample basis. For convergence, it depends on our loss tolerance by the termination of the procedure. SGD may exhibit oscillating behavior in later epochs, whereas GD's complete information of the gradient allows it to converge directly. So with these reservations, I would choose SGD for low run time, GD for quicker convergence.

6. I have also experimented with mini-batch SGD. The plot below is done with parameters $\lambda = 10e^{-2}, \eta = 0.1/t$ over 100 iterations/epochs, the objective function is ridge regression:




With a timeit module, corresponding batch sizes have the following times (for 100 epochs):
batch size = 2: 0.02902 sec
batch size = 5: 0.04003 sec
batch size = 10: 0.06504 sec
batch size = 20: 0.12308 sec

As we can see, setting the batch size is a trade-off between number of iterations until convergence versus run-time per batch. If we have large amount of samples, it would be better to choose a smaller batch size.

# 3 Risk Minimization

## 3.1 Square Loss

1. Let $y$ be a random variable with a known distribution, and consider the square loss function $\ell(a, y) = (a - y)^2$. We want to find the action $a^*$ that has minimal risk. That is, we want to find $a^* = \arg\min_a \mathbb{E}\,(a - y)^2$, where the expectation is with respect to $y$. Show that $a^* = \mathbb{E}y$, and the Bayes risk (i.e. the risk of $a^*$) is $\mathrm{Var}(y)$. In other words, if you want to try to predict the value of a random variable drawn, the best you can do (for minimizing square loss) is to predict the mean of the distribution. Your expected loss for predicting the mean will be the variance of the distribution. [Hint: Recall that $\mathrm{Var}(y) = \mathbb{E}y^2 - (\mathbb{E}y)^2$.]

$$\begin{aligned}\mathbb{E}\,() = \mathbb{E}\left((\mathbb{E}y - y)^2\right) &= \mathbb{E}\left((\mathbb{E}y)^2 - 2y\mathbb{E}y + y^2\right) \\ &= (\mathbb{E}y)^2 - 2\mathbb{E}y\mathbb{E}y + \mathbb{E}(y^2) \\ &= \mathbb{E}(y^2) - (\mathbb{E}y)^2 = \mathrm{Var}(y)\end{aligned}$$

We have shown that the risk function with action $\mathbb{E}y$ yields $\mathrm{Var}(y)$, our target variable's variance that cannot be reduced by any model (irreducible error). In other words, for the possible family of

actions $A$, $\forall a \in A, \mathbb{E}l(a^*, y) \leq \mathbb{E}l(a, y)$. The first statement follows. This then implies $\text{Var}(y)$ is the Bayes risk, the risk of $a^*$.

2. (a) I show that $f(X = x) = \mathbb{E}(y|x)$ is the desired decision function.

$$\begin{aligned} R(f) = \mathbb{E}\left((y - f(x))^2|x\right) &= \mathbb{E}\left((y - \mathbb{E}(y|x))^2|x\right) \\ &= \text{Var}(y|x) \end{aligned}$$

Our irreducible error is the variance of the test target given information about $x$, $\text{Var}(y|x)$. To see $f$ has the desired property, suppose $\exists g \in F$ s.t. $R(g) < R(f) = \text{Var}(y|x)$. This implies some better performing $g \in F$ with action $g(x)$ has less expected error than $\text{Var}(y|x)$, which is impossible.

(b) Draw a set $S$ of sample, target variable tuples $(x_i, y_i)$ indexed by $i$ from $P_{X \times Y}$. Since expectation is the average square loss and $f^*(x) = \mathbb{E}(y|x) = \text{argmin}_a \mathbb{E}((a - y)^2|x)$ for any $x$, the following inequality holds:

$$\begin{aligned} \mathbb{E}\left[(f^*(x_i) - y_i)^2 \mid S\right] &= \frac{E((f^*(x_i) - y_i)^2)}{P(S)} \\ &\leq \frac{E((f(x_i) - y_i)^2)}{P(S)} = \mathbb{E}\left[(f(x_i) - y_i)^2 \mid S\right] \end{aligned}$$

Where $f$ is any decision function. Continue adding tuples to $S$ until $S = X$, we see that the denominator equals one, and what remains is the desired inequality.

## 3.2 [Optional] Median Loss

1. (Optional) Show that for the absolute loss $\ell(\hat{y}, y) = |y - \hat{y}|$, then $f^*(x)$ is a Bayes decision function iff $f^*(x)$ is the median of the conditional distribution of $y$ given $x$. [Hint: As in the previous section, consider one $x$ at time. It may help to 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 to "quantile regression" for any chosen quantile.