

# Machine Learning and Computational Statistics, Spring 2017

## Homework 1: GD, SGD, and Ridge Regression

**Due: Sunday, February 5, 2017, at 10pm (Submit via GradeScope)**

**Instructions:** Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.  $\text{\LaTeX}$ ,  $\text{\LyX}$ , or MathJax via iPython), though if you need to you may scan handwritten work. You may find the `minted` package convenient for including source code in your  $\text{\LaTeX}$  document.

### 1 Introduction

In this homework you will implement ridge regression using gradient descent and stochastic gradient descent. We've provided a lot of support Python code to get you started on the right track. References below to particular functions that you should modify are referring to the support code, which you can download from the website. If you have time after completing the assignment, you might pursue some of the following:

- Study up on numpy's `broadcasting` to see if you can simplify and/or speed up your code.
- Think about how you could make the code more modular so that you could easily try different loss functions and step size methods.
- Experiment with more sophisticated approaches to setting the step sizes for SGD (e.g. try out the recommendations in “Bottou’s SGD Tricks” on the website)
- Instead of taking 1 data point at a time, as in SGD, try minibatch gradient descent, where you use multiple points at a time to get your step direction. How does this effect convergence speed? Are you getting computational speedup as well by using vectorized code?
- Advanced: What kind of loss function will give us “quantile regression”?

I encourage you to develop the habit of asking “what if?” questions and then seeking the answers. I guarantee this will give you a much deeper understanding of the material (and likely better performance on the exam and job interviews, if that’s your focus). You’re also encouraged to post your interesting questions on Piazza under “questions”, or on CrossValidated (<http://stats.stackexchange.com/>).

## 2 Linear Regression

### 2.1 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 (introduced in a later problem), 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 in the training set are in  $[0, 1]$ . Each feature gets its own transformation. We then apply the same transformations to each feature on the test<sup>1</sup> 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?)

Solution:

```
def feature_normalization(train, test):
    """
    Rescale the data so that each feature in the training set is in the
    interval [0,1], and apply the same transformations to the test
    set, using the statistics computed on the training set.
    Args:
        train – training set, a 2D numpy array of size (numinstances,
            numfeatures)
        test – test set, a 2D numpy array of size (numinstances,
            numfeatures)
    Returns:
        train_normalized – training set after normalization
        test_normalized – test set after normalization
    """
    # TODO
    # Get the stats
    train_max = train.max(axis = 0)
    train_min = train.min(axis = 0)

    # Delete the features that have constant value
    equal_indicator = (train_max != train_min)
    train = train[:, equal_indicator]
    test = test[:, equal_indicator]
    train_max = train_max[equal_indicator]
    train_min = train_min[equal_indicator]
```

---

<sup>1</sup>Throughout this assignment we refer to the “test” set. It may be more appropriate to call this set the “validation” set, as it will be a set of data on which compare the performance of multiple models. Typically a test set is only used once, to assess the performance of the model that performed best on the validation set.

```
# Normalize (uses array broadcasting http://wiki.scipy.org/EricksBroadcastingDoc)
train_normalized = (train - train_min) / (train_max - train_min)
test_normalized = (test - train_min) / (train_max - train_min)

return train_normalized, test_normalized
```

## 2.2 Gradient Descent Setup

In linear regression, we consider the hypothesis space of linear functions  $h_\theta : \mathbf{R}^d \rightarrow \mathbf{R}$ , where

$$h_\theta(x) = \theta^T x,$$

for  $\theta, x \in \mathbf{R}^d$ , and we choose  $\theta$  that minimizes the following “square loss” objective function:

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

where  $(x_1, y_1), \dots, (x_m, y_m) \in \mathbf{R}^d \times \mathbf{R}$  is our training data.

While this formulation of linear regression is very convenient, it’s more standard to use a hypothesis space of “affine” functions:

$$h_{\theta,b}(x) = \theta^T x + b,$$

which allows a “bias” or nonzero intercept term. The standard way to achieve this, while still maintaining the convenience of the first representation, is to add an extra dimension to  $x$  that is always a fixed value, such as 1. You should convince yourself that this is equivalent. We’ll assume this representation, and thus we’ll actually take  $\theta, x \in \mathbf{R}^{d+1}$ .

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  $y = (y_1, \dots, y_m)^T \in \mathbf{R}^{m \times 1}$  be a the “response”. Write the objective function  $J(\theta)$  as a matrix/vector expression, without using an explicit summation sign.  
Solution:  $\frac{1}{2m} (X\theta - y)^T (X\theta - y)$
2. Write down an expression for the gradient of  $J$ .  
Solution:  $\nabla_\theta J(\theta) = \frac{1}{m} X^T (X\theta - y)$
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.]  
Solution:  $J(\theta + \eta\Delta) - J(\theta) \approx \nabla_\theta J(\theta)^T (\theta + \eta\Delta - \theta) = \nabla_\theta J(\theta)^T \eta\Delta$
4. Write down the expression for updating  $\theta$  in the gradient descent algorithm. Let  $\eta$  be the step size.  
Solution:  $\theta_{n+1} = \theta_n - \eta \nabla_{\theta_n} J(\theta_n)$ , where  $\eta > 0$

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.

Solution:

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

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.

Solution:

```
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 array of size (
            num_features)
    Returns:
        grad - gradient vector, 1D numpy array of size (
            num_features)
    """
    # TODO
    num_instances = y.shape[0]
```

```
grad = 1.0/num_instances * np.dot((np.dot(X, theta) - y), X)
return grad
```

## 2.3 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  $J : \mathbf{R}^d \rightarrow \mathbf{R}$  is differentiable, then for any direction vector  $\Delta \in \mathbf{R}^d$ , the directional derivative of  $J$  at  $\theta$  in the direction  $\Delta$  is given by<sup>2</sup>

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

We can approximate this directional 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. In other words, take  $\Delta = (1, 0, 0, \dots, 0)$  to get the first component of the gradient. Then take  $\Delta = (0, 1, 0, \dots, 0)$  to get the second component. And so on. See [http://ufldl.stanford.edu/wiki/index.php/Gradient\\_checking\\_and\\_advanced\\_optimization](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. 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.

Solution:

```
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
    """Implement Gradient Checker
    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
```

---

<sup>2</sup>Of course, it is also given by the more standard definition of directional derivative,  $\lim_{\varepsilon \rightarrow 0} \frac{1}{\varepsilon} [J(\theta + \varepsilon \Delta) - J(\theta)]$ . The form given gives a better approximation to the derivative when we are using small (but not infinitesimally small)  $\varepsilon$ .

```

num_features = theta.shape[0]
approx_grad = np.zeros(num_features) #Initialize the gradient
we approximate
#TODO
for i in range(num_features):
    # Set the direction vector for the directional derivative
    direction = np.zeros(num_features)
    direction[i] = 1
    # Compute the approximate directional derivative in the
    chosen direction
    approx_grad[i] = (compute_square_loss(X, y, theta+epsilon*
        direction) - compute_square_loss(X, y, theta-epsilon*
        direction))/(2*epsilon)
#END TODO
error = np.linalg.norm(true_gradient - approx_grad)
return (error < tolerance)

def generic_gradient_checker(X, y, theta, objective_func,
    gradient_func, \ epsilon=0.01, tolerance=1e-4):
    true_gradient = gradient_func(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):
        direction = np.zeros(num_features)
        direction[i] = 1
        approx_grad[i] = (objective_func(X, y, theta+epsilon*
            direction) - objective_func(X, y, theta-epsilon*
            direction))/(2*epsilon)

    return np.linalg.norm(true_gradient - approx_grad) < epsilon

```

## 2.4 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`.

Solution:

```

def batch_grad_descent(X, y, alpha=0.1, num_iter=1000,
    grad_checking=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
numIter – number of iterations to run
grad_checker – a boolean value indicating whether checking
    the gradient
Returns:
    theta_hist – the history of parameter vector,
    loss_hist – the history of loss function, 1D numpy array
"""
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)
theta = np.ones(num_features) #initialize theta

#TODO
for i in range(num_iter):
    theta_hist[i] = theta
    loss_hist[i] = compute_square_loss(X, y, theta)
    grad = compute_square_loss_gradient(X, y, theta) #compute
        the gradient
    if grad_checker:
        if not grad_checker(X, y, theta):
            sys.exit("Wrong gradient")
    if alpha == "stepsize_search":
        step_size = stepsize_search(X, y, theta,
            compute_square_loss, compute_square_loss_gradient)
        theta = theta - step_size*grad
    else:
        theta = theta - alpha*grad #update theta
    theta_hist[i+1] = theta
    loss_hist[i+1] = compute_square_loss(X, y, theta)
return theta_hist, loss_hist

```

2. Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge<sup>3</sup>. 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.

---

<sup>3</sup>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](https://www.cs.cmu.edu/~ggordon/10725-F12/scribes/10725_Lecture5.pdf), Theorem 5.1.

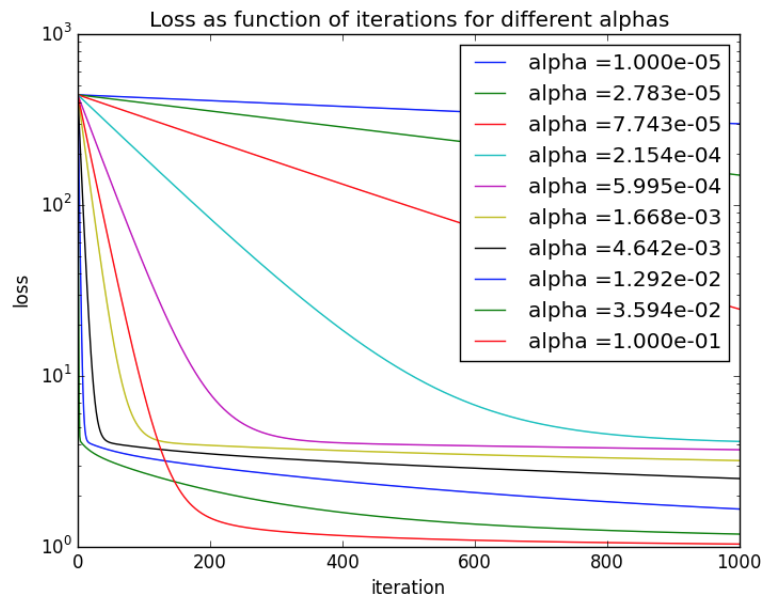


Figure 1:

(a) Solution:

$\alpha = 0.5$  diverges.

3. (Optional, but recommended) 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.)

(a) Solution:

```
def stepsize_search(X, y, theta, loss_func, grad_func, epsilon
=1e-6):
    alpha = 1.0
    gamma = 0.5
    loss = loss_func(X, y, theta)
    gradient = grad_func(X, y, theta)
    while True:
        theta_next = theta - alpha*grad_func(X,y, theta)
        loss_next = loss_func(X, y, theta_next)
        if loss_next > loss-epsilon:
            alpha = alpha*gamma
        else:
            return alpha
```



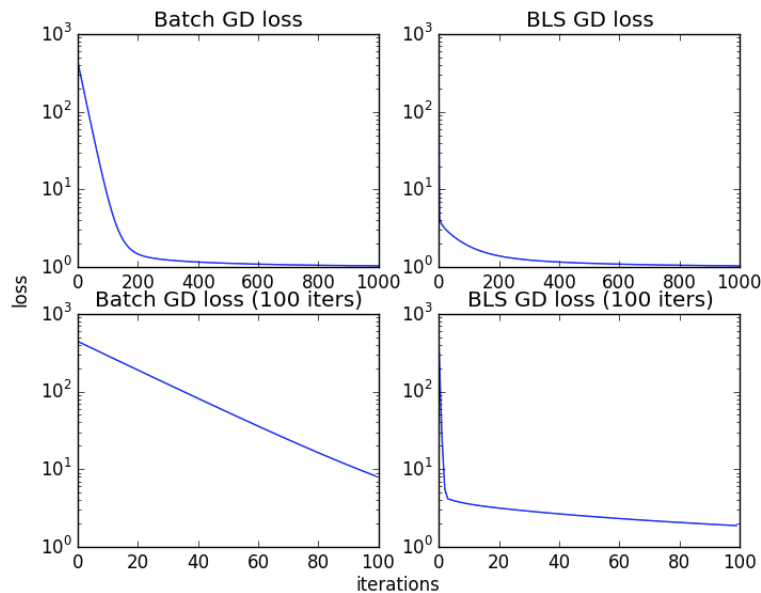


Figure 2:  
BLS = Backtracking Line Search

Backtrack line search converges in less iterations (see two lower subplots)- however, it does not yield a significantly different loss after 1000 iterations.

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

When we have a large number of features compared to instances, regularization can help control overfitting. Ridge regression is linear regression with  $L_2$  regularization. The regularization term is sometimes called a penalty term. The objective function for ridge regression is

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

where  $\lambda$  is the regularization parameter, which controls the degree of regularization. Note that the bias parameter is being regularized as well. We will address that below.

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

Solution:

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

2. Implement `compute_regularized_square_loss_gradient`.

Solution:

```

def compute_regularized_square_loss_gradient(X, y, theta,
lambda_reg):
    num_instances = y.shape[0]
    grad = 1.0/num_instances * np.dot((np.dot(X, theta.T) - y), X)
        + 2*lambda_reg*theta
    return grad

```

3. Implement `regularized_grad_descent`.

Solution:

```

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 = 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)
    for i in range(num_iter):
        theta_hist[i] = theta
        loss_hist[i] = compute_square_loss(X, y, theta)
        grad = compute_regularized_square_loss_gradient(X, y, theta
            , lambda_reg)
        theta = theta - alpha*grad

    theta_hist[i+1] = theta
    loss_hist[i+1] = compute_square_loss(X, y, theta)
    return theta_hist, loss_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).

Solution:

Suppose we have an affine function  $f(x) = \theta_0 B + \theta^T x$ , where we've separated out  $B$  from the vector  $x$ . If we increase  $B$ , the corresponding coefficient  $\theta_0$  will have to decrease by the same factor to end up with the same function  $f$ . A smaller coefficient  $\theta_0$  incurs less regularization penalty. As  $B \rightarrow \infty$ ,  $\beta_0 \rightarrow 0$  and the regularization effect approaches 0.

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

Solution:

For any set of data  $(x_1, y_1), \dots, (x_n, y_n) \in \mathbf{R}^d \times \mathbf{R}$ , loss function  $\ell : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$  and

hyperparameters  $B, \lambda_1, \lambda_2 > 0$ , let

$$(\theta_{1,B}^*, \theta_{2,B}^*) = \arg \min_{\theta_1 \in \mathbf{R}, \theta_2 \in \mathbf{R}^d} \sum_{i=1}^n \ell(\theta_1 B + \theta_2^T x_i, y_i) + \lambda_1 \theta_1^2 + \lambda_2 \|\theta_2\|_2^2.$$

Note that this is ridge regression where we have used  $B > 0$  rather than the usual 1 as our “bias feature”. We have also separated out the regularization of the bias parameter  $\theta_1 \in \mathbf{R}$  from the rest of the parameters  $\theta_2 \in \mathbf{R}^d$ . The corresponding prediction function is  $h_B(x) = \theta_{1,B}^* B + (\theta_{2,B}^*)^T x$ .

Let us also consider the following formulation of ridge regression, in which we use the usual 1 as the bias feature, but we scale the regularization on the bias term by  $\frac{1}{B^2}$ :

$$(\theta_{1,1}^*, \theta_{2,1}^*) = \arg \min_{\theta_1, \theta_2} \sum_{i=1}^n \ell(\theta_1 + \theta_2^T x_i, y_i) + \frac{\lambda_1}{B^2} \theta_1^2 + \lambda_2 \|\theta_2\|_2^2.$$

The corresponding prediction function is  $h_1(x) = \theta_{1,1}^* + (\theta_{2,1}^*)^T x$ . Below we will show that  $h_1 = h_B$ .

In words, this result says that using  $B$  as the “bias feature” and using  $\lambda_1 \theta_1^2$  as our penalty for the bias parameter is equivalent to using 1 as our bias feature, and penalizing  $\theta_1$  by  $\lambda_1 \theta_1^2 / B^2$ . So, for example, if we use a very large  $B$  as our bias feature, it’s equivalent to having a very small penalty on  $\theta_1$  when we use 1 as our bias feature. If  $B$  is very small, it’s equivalent to using 1 as our bias feature but having a very large penalty.

**Proof:**

Let’s define our family of objective functions as

$$H(\theta_1, \theta_2, B, \lambda_1, \lambda_2) = \sum_{i=1}^n \ell(\theta_1 B + \theta_2^T x_i, y_i) + \lambda_1 \theta_1^2 + \lambda_2 \|\theta_2\|_2^2,$$

where  $\lambda_1, \lambda_2 > 0$ .

So we want to show that if

$$(\theta_{1,B}^*, \theta_{2,B}^*) = \arg \min_{\theta_1 \in \mathbf{R}, \theta_2 \in \mathbf{R}^d} H(\theta_1, \theta_2, B, \lambda_1, \lambda_2)$$

and

$$(\theta_{1,1}^*, \theta_{2,1}^*) = \arg \min_{\theta_1 \in \mathbf{R}, \theta_2 \in \mathbf{R}^d} H(\theta_1, \theta_2, 1, \lambda_1/B^2, \lambda_2)$$

then  $h_1(x) = \theta_{1,1}^* + (\theta_{2,1}^*)^T x$  and  $h_B(x) = \theta_{1,B}^* B + (\theta_{2,B}^*)^T x$  are equal as functions.

Note that

$$\begin{aligned} H(\theta_1, \theta_2, B, \lambda_1, \lambda_2) &= \sum_{i=1}^n \ell(\theta_1 B + \theta_2^T x_i, y_i) + \lambda_1 \theta_1^2 + \lambda_2 \|\theta_2\|_2^2 \\ &= \sum_{i=1}^n \ell(\theta_1 B + \theta_2^T x_i, y_i) + \frac{\lambda_1}{B^2} (\theta_1^2 B^2) + \lambda_2 \|\theta_2\|_2^2 \\ &= H(\theta_1 B, \theta_2, 1, \lambda_1/B^2, \lambda_2) \end{aligned}$$

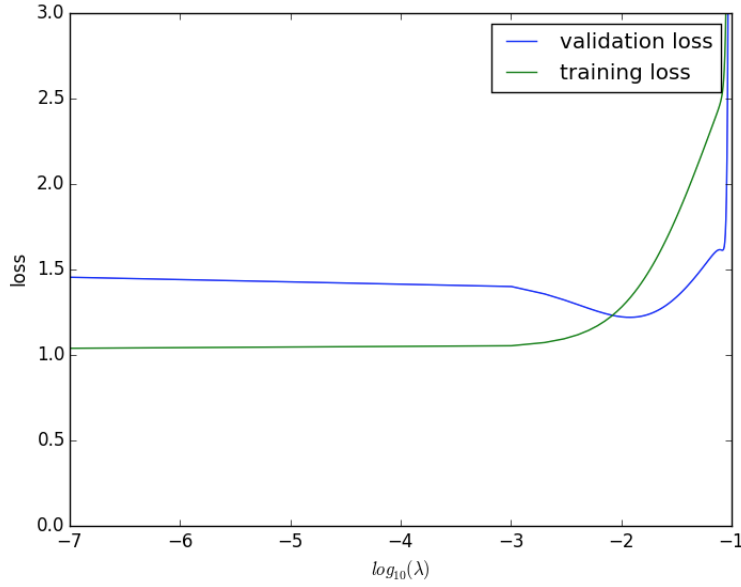


Figure 3:

So since

$$\begin{aligned}
 (\theta_{1,B}^*, \theta_{2,B}^*) &= \arg \min_{\theta_1 \in \mathbf{R}, \theta_2 \in \mathbf{R}^d} H(\theta_1, \theta_2, B, \lambda_1, \lambda_2) \\
 &= \arg \min_{\theta_1 \in \mathbf{R}, \theta_2 \in \mathbf{R}^d} H(\theta_1 B, \theta_2, 1, \lambda_1/B^2, \lambda_2)
 \end{aligned}$$

and

$$(\theta_{1,1}^*, \theta_{2,1}^*) = \arg \min_{\theta_1 \in \mathbf{R}, \theta_2 \in \mathbf{R}^d} H(\theta_1, \theta_2, 1, \lambda_1/B^2, \lambda_2),$$

then clearly  $\theta_{1,1}^* = \theta_{1,B}^* B$  and  $\theta_{2,B}^* = \theta_{2,1}^*$ . Thus  $h_1 = h_B$ .

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  $\lambda$ .

(a) Solution:

The optimal  $\lambda$  is around  $10^{-2}$ .

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

Solution:

Choose the  $\theta$  that minimizes the test error.

## 2.6 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. Before we begin cycling through the training examples, it is important to shuffle the examples into a random order. You can use the same ordering for each epoch, though optionally you could investigate whether reshuffling after each epoch speeds up convergence.

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

Solution:

$$\theta_i = \theta_{i-1} - \eta [x_{i-1}(\theta_{i-1}^T x_{i-1} - y_{i-1}) + 2\lambda\theta_{i-1}]$$

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.)

Solution:

```
def stochastic_grad_descent(X, y, alpha=0.1, lambda_reg=1, num_iter
=1000):
    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, num_features))
        #Initialize theta_hist
    loss_hist = np.zeros((num_iter, num_instances))
    pointList = range(num_instances)

    #set stepsize
    if isinstance(alpha, float):
        step_size = alpha
    else:
        step_size_p = 1

    for i in range(num_iter):
        #shuffle the points
        #change alpha
        np.random.shuffle(pointList)
        for j in range(num_instances):
```

```

#store the historical theta
theta_hist[i, j] = theta
loss_hist[i, j] = compute_square_loss(X, y, theta)
#simultaneously update theta
grad = (np.dot(X[j], theta.T) - y[j])*X[j]+2*lambda_reg
      *theta
if isinstance(alpha, str):
    if alpha == "1/t":
        theta = theta - 1./step_size_p * grad
    elif alpha == "1/sqrt(t)":
        theta = theta - 1./np.sqrt(step_size_p) * grad
        step_size_p+=1
else:
    theta = theta - alpha*grad # update theta

return theta_hist, loss_hist

```

- Use SGD to find  $\theta_\lambda^*$  that minimizes the ridge regression objective for the  $\lambda$  and  $B$  that you selected in the previous problem. (If you could not solve the previous problem, choose  $\lambda = 10^{-2}$  and  $B = 1$ ). 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 the 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 (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? Two things to note: 1) In this case we are investigating the convergence rate of the optimization algorithm with different step size schedules, thus we're interested in the value of the objective function, which includes the regularization term. 2) As we'll learn in an upcoming lecture, SGD convergence is much slower than GD once we get close to the minimizer. (Remember, the SGD step directions are very noisy versions of the GD step direction). If you look at the objective function values on a logarithmic scale, it may look like SGD will never find objective values that are as low as GD gets. In terms we'll discuss in Week 2, GD has much smaller "optimization error" than SGD. However, this difference in optimization error is usually dominated by other sources of error (estimation error and approximation error, which we'll also discuss in Week 2). Moreover, for very large datasets, SGD (or minibatch GD) is much faster (by wall-clock time) than GD to reach a point that's close enough to the minimum.

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

Solution:

About  $10^{-3}$  seconds

- 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?

Solution:

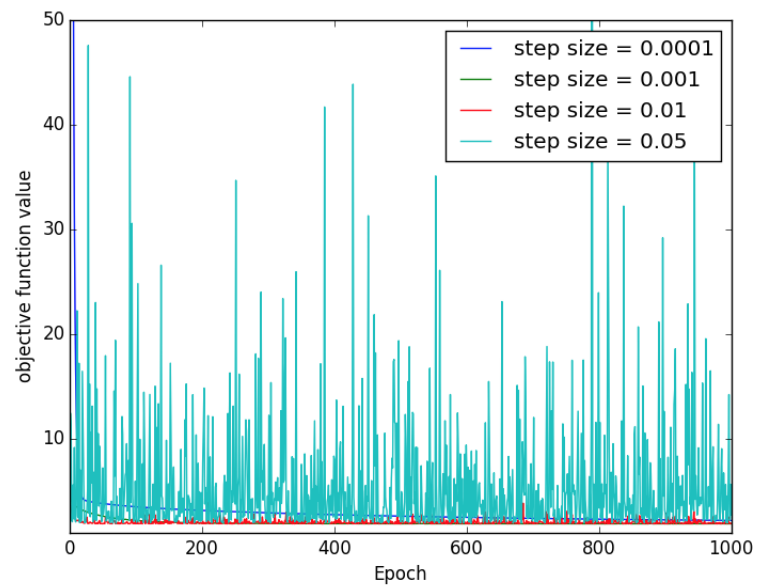


Figure 4:

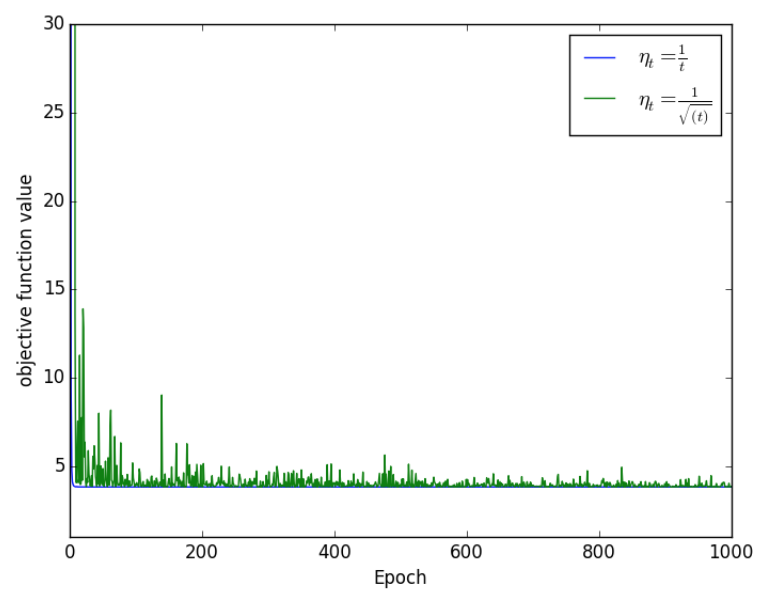


Figure 5:

For this dataset, the running time for SGD took longer than GD, but it took fewer epochs to converge for SGD.

### 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  $\text{Var}(y)$ . 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  $\text{Var}(y) = \mathbb{E}y^2 - (\mathbb{E}y)^2$ .]

$$\begin{aligned} \mathbb{E}\ell(a, y) &= \mathbb{E}(a - y)^2 \\ &= \mathbb{E}[a^2 - 2ay + y^2] \\ &= a^2 - 2a\mathbb{E}y + \mathbb{E}y^2 \end{aligned}$$

One can find the minimum of this quadratic by basic algebra or calculus. First order condition gives  $2a - 2\mathbb{E}y = 0 \implies a^* = \mathbb{E}y$ . Plugging back in this minimizer, we find that  $\mathbb{E}\ell(a^*, y) = (\mathbb{E}y)^2 - 2(\mathbb{E}y)^2 + \mathbb{E}y^2 = \text{Var}(y)$ .

2. Now let's introduce an input. Recall that the **expected loss** or “**risk**” of a decision function  $f : \mathcal{X} \rightarrow \mathcal{A}$  is

$$R(f) = \mathbb{E}\ell(f(x), y),$$

where  $(x, y) \sim P_{\mathcal{X} \times \mathcal{Y}}$ , and the **Bayes decision function**  $f^* : \mathcal{X} \rightarrow \mathcal{A}$  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  $\mathcal{A} = \mathcal{Y} = \mathbf{R}$ . We will show for the square loss  $\ell(a, y) = (a - y)^2$ , the Bayes decision function is  $f^*(x) = \mathbb{E}[y \mid x]$ , where the expectation is over  $y$ . As before, we assume know the data-generating distribution  $P_{\mathcal{X} \times \mathcal{Y}}$ .

- (a) We'll approach this problem by finding the optimal action for any given  $x$ . If somebody tells us  $x$ , we know that the corresponding  $y$  is coming from the conditional distribution  $y \mid x$ . For a particular  $x$ , what value should we predict (i.e. what action  $a$  should we produce) that has minimal expected loss? Express your answer as a decision function  $f(x)$ , which gives the best action for any given  $x$ . In mathematical notation, we're looking for  $f^*(x) = \arg \min_a \mathbb{E}[(a - y)^2 \mid x]$ , where the expectation is with respect to  $y$ . Solution: Based on the previous problem, it's  $f^*(x) = \mathbb{E}[y \mid x]$ .
- (b) In the previous problem we produced a decision function  $f^*(x)$  that minimized the risk for each  $x$ . In other words, for any other decision function  $f(x)$ ,  $f^*(x)$  is going to be at least as good as  $f(x)$ , for every single  $x$ . That is

$$\mathbb{E}[(f^*(x) - y)^2 \mid x] \leq \mathbb{E}[(f(x) - y)^2 \mid x],$$



for all  $x$ . To show that  $f^*(x)$  is the Bayes decision function, we need to show that

$$\mathbb{E} \left[ (f^*(x) - y)^2 \right] \leq \mathbb{E} \left[ (f(x) - y)^2 \right]$$

for any  $f$ . Explain why this is true. Solution: In the expression  $\mathbb{E} \left[ (f^*(x) - y)^2 \mid x \right] \leq \mathbb{E} \left[ (f(x) - y)^2 \mid x \right]$ , each side of the inequality is a [deterministic] function of  $x$ . Let's call the left side  $g(x)$  and the right side  $h(x)$ . So  $g(x) \leq h(x)$  for all  $x$ . Then it's certainly the case that  $\mathbb{E}g(x) \leq \mathbb{E}h(x)$ . Finally we just used the law of iterated expectations, which says that  $\mathbb{E}(\mathbb{E}[y \mid x]) = \mathbb{E}y$ .

### 3.2 [Optional] Median Loss

- (Optional) Show that for the absolute loss  $\ell(\hat{y}, y) = |y - \hat{y}|$ , then  $f^*(x)$  is a Bayes decision function if  $f^*(x)$  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.

(a) Solution:

Let  $m = \text{median}(Y \mid X = x)$ . We'll compare the risk for predicting  $m$  to the risk for predicting some other value  $b$ . The calculation will be done for two parts:  $m < b$  and  $m > b$ .

**Suppose  $m < b$ .**

If  $Y \leq m$ , then

$$|Y - b| - |Y - m| = (b - Y) - (m - Y) = b - m.$$

If  $Y > m$ , then

$$\begin{aligned} |Y - b| - |Y - m| &= |Y - b| - (Y - m) \\ &\geq (Y - b) - (Y - m) \\ &= m - b. \end{aligned}$$

Putting this together, we get:

$$\begin{aligned} |Y - b| - |Y - m| &\geq (b - m) 1(Y \leq m) - (b - m) 1(Y > m) \\ &= (b - m) [1(Y \leq m) - 1(Y > m)] \end{aligned}$$

Taking expectations yields

$$\begin{aligned} \mathbb{E}(|Y - b| - |Y - m|) &\geq \underbrace{(b - m)}_{>0} \underbrace{\{\mathbb{P}(Y \leq m) - \mathbb{P}(Y > m)\}}_{\geq 0 \text{ by definition of median}} \\ &\geq 0 \end{aligned}$$

**For  $m > b$ :**

If  $Y \geq m$ , then

$$\begin{aligned} |Y - b| - |Y - m| &= Y - b - (Y - m) \\ &= m - b, \end{aligned}$$

If  $Y < m$

$$\begin{aligned} |Y - b| - |Y - m| &= |Y - b| - (m - Y) \\ &\geq b - Y - (m - Y) \\ &= b - m. \end{aligned}$$

Putting this together, we get:

$$\begin{aligned} |Y - b| - |Y - m| &\geq (m - b) 1(Y \geq m) - (m - b) 1(Y < m) \\ &= (m - b) [1(Y \geq m) - 1(Y < m)] \end{aligned}$$

Taking expectations yields

$$\begin{aligned} \mathbb{E}(|Y - b| - |Y - m|) &\geq \underbrace{(m - b)}_{>0} \underbrace{\{\mathbb{P}(Y \geq m) - \mathbb{P}(Y < m)\}}_{\geq 0 \text{ by definition of median}} \\ &\geq 0 \end{aligned}$$

So for any values of  $m$  and  $b$ ,  $\mathbb{E}|Y - b| \geq \mathbb{E}|Y - m|$ .