Machine Learning and Computational Statistics, Spring 2016
Homework 1: Ridge Regression and SGD

Due: Thursday, February 2, 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. LaTeX, 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 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[1] 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):

    max_train = np.zeros(train.shape[1])
    min_train = np.zeros(train.shape[1])

    train_normalized = np.zeros_like(train)
    test_normalized = np.zeros_like(test)

    max_train = np.amax(train, 0)
    min_train = np.amin(train, 0)

    train_normalized = (train - min_train)/(max_train - min_train)
    test_normalized  = (test - min_train)/(max_train - min_train)


    return(train_normalized, test_normalized)
```

[1] 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.

2.2 Gradient Descent Setup

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

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

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

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

where $(x_1, y_1), \ldots, (x_m, y_m) \in \mathbf{R}^d \times \mathbf{R}$ 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 $y = (y_1, \ldots, y_m)^T \in \mathbf{R}^{m \times 1}$ be a the "response". Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign.

$$J(\theta) = \frac{1}{2m} \left(\begin{bmatrix} x_{11} & x_{12} & x_{13} & \ldots & x_{1d} & 1 \\ x_{21} & x_{22} & x_{23} & \ldots & x_{2d} & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & 1 \\ x_{m1} & x_{m2} & x_{m3} & \ldots & x_{md} & 1 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_d \\ b \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \right)^2$$

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

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

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

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 $J(\theta + \eta\Delta) - J(\theta)$. [This approximation is called a "linear" or "first-order" approximation.]

$$J(\theta + \eta\Delta) - J(\theta) = \nabla J(\theta) \eta \Delta$$

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

$$\theta_{\mathrm{i}\,+\,1} = \theta_i - \eta \nabla J(\theta_i)$$

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

    loss = 0 #initialize the square_loss
    loss = (np.dot(X, theta) - y)
    return (np.dot(loss.T, loss)) / (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(X, y, theta):

    grad = np.dot(X.T, np.dot(X, theta) - y)/X.shape[0]
    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 $J : \mathbf{R}^d \to \mathbf{R}$ 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[2]

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

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 $\Delta = (1, 0, 0, \ldots, 0)$ to get the first component of the gradient. Then take $\Delta = (0, 1, 0, \ldots, 0)$ to get the second component. And so on. 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. 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 grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):

    true_gradient = compute_square_loss_gradient(X, y, theta) #the true gradient
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we approximate
    e = np.zeros(num_features)

    for i in range(num_features):
        e[i] = 1
        approx_grad[i] = (compute_square_loss(X, y, theta + epsilon * e) -
        compute_square_loss(X, y, theta - epsilon * e))/(2 * epsilon)
        e[i] = 0

    return np.allclose(approx_grad, true_gradient, atol = tolerance)
```

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

```
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 = theheta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we approximate

    e = np.zeros(num_features)

    for i in range(num_features):
        e[i] = 1
        approx_grad[i] = (objective_func(X, y, theta + epsilon * e) -
        objective_func(X, y, theta - epsilon * e))/(2 * epsilon)
        e[i] = 0

    dist = np.linalg.norm(true_gradient - approx_grad)

    return np.allclose(approx_grad, true_gradient, atol = tolerance)
```

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`.

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

    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(num_iter):

        grad = compute_square_loss_gradient(X, y, theta)
        theta = theta - (alpha * grad)  #alpha = 0.093
        loss_hist[i + 1] = compute_square_loss(X, y, theta)
        theta_hist[i + 1] = theta

    return(loss_hist, theta_hist)
```

2. Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge[3]. 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 all values of step-size > 0.1, the gradient descent diverges and the diversion becomes more and more sharp as the step-size increases.

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.)

[3]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.

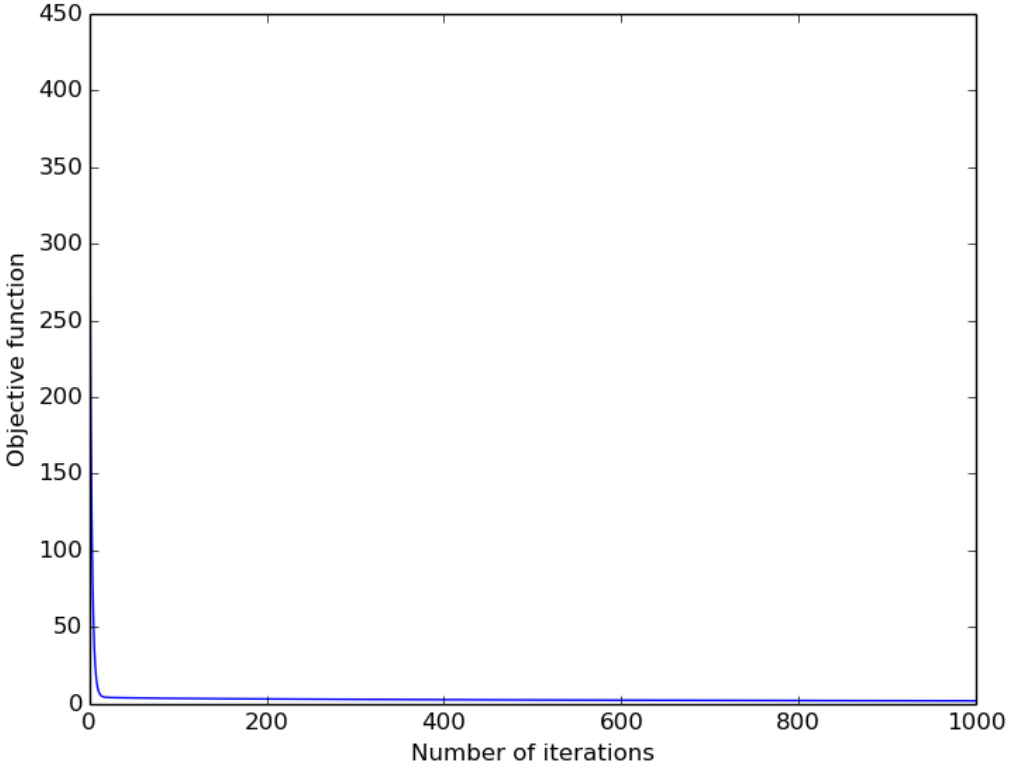

Figure 1: Objective Function v/s No of steps with $\eta = 0.01$

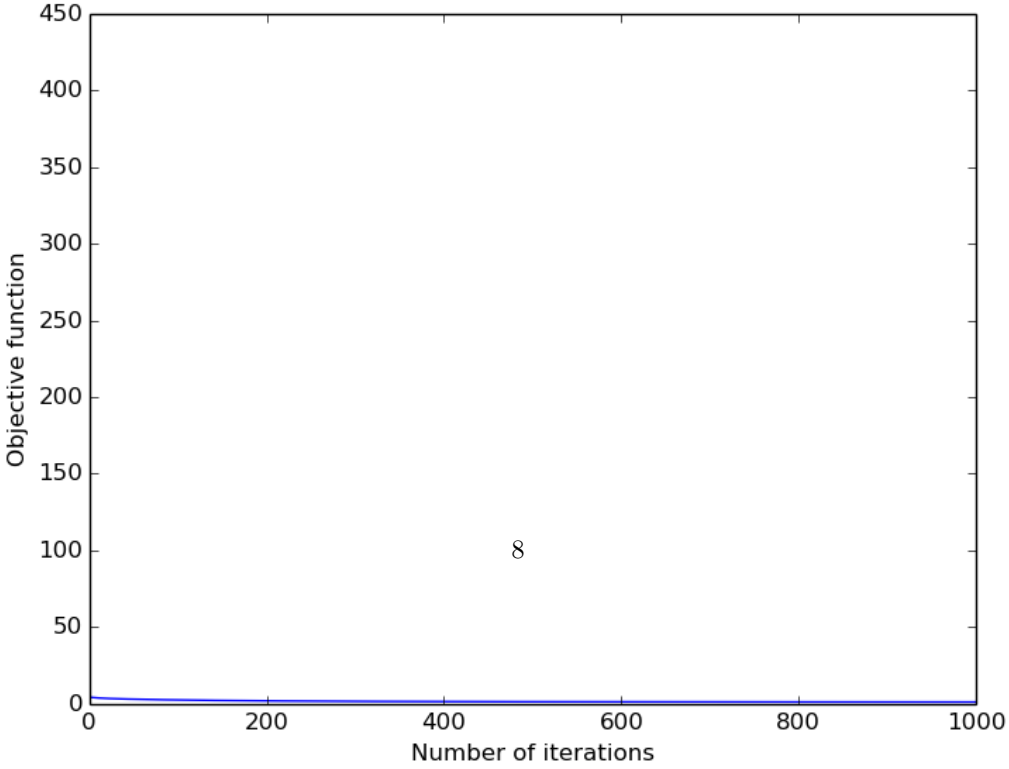

Figure 2: Objective Function v/s No of steps with $\eta = 0.05$

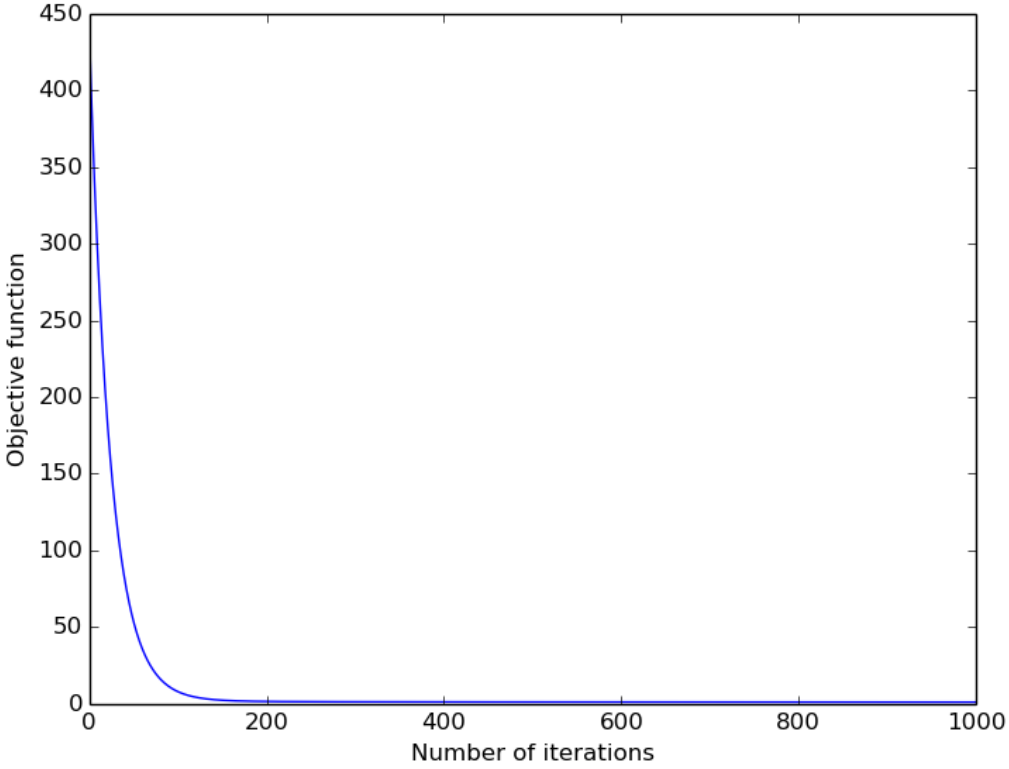

Figure 3: Objective Function v/s No of steps with $\eta = 0.1$

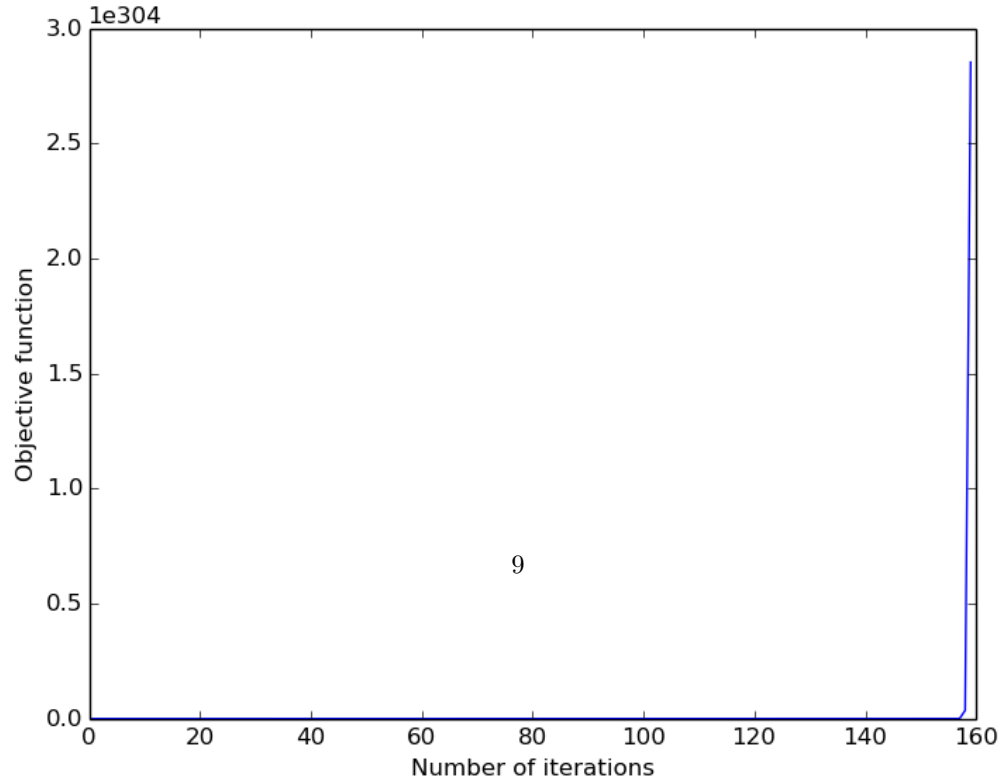

Figure 4: Objective Function v/s No of steps with $\eta = 0.5$

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 λ 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 θ in the gradient descent algorithm.

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

2. Implement `compute_regularized_square_loss_gradient`.

```
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):

    grad = (np.dot(X.T, np.dot(X, theta) - y))/X.shape[0] + 2 * lambda_reg * theta
    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
    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) #Initialize loss_hist

    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta)

    for i in range(num_iter):

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

        theta = theta - (alpha * grad)  #alpha = 0.093

        loss_hist[i + 1] = compute_square_loss(X, y, theta)
        loss_hist[i + 1] = loss_hist[i + 1] + lambda_reg * np.dot(theta.T, theta)
        theta_hist[i + 1] = 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).

   Regularization is used to prevent overfitting. In linear regression : $y = w^T x + b$ overfitting is caused by the vector w when it becomes too complex and adapted to the training data. Hence for the problem we require bias, regularization leads to minimization of bias term as well, which is not required to prevent overfitting. We don't want to regularize the bias as it may reduce the bias term which might be crucial for some classification problem.

   When we change the extra dimension in the x vector to B which has a large value, the coefficient of the bias term in ridge regression expression is directly proportional to the sum of B and λ. Where λ is the penalty parameter for regularization and as we increase B the regularization on bias becomes weaker as the coefficient of the bias term tends to be almost constant. However, since the coefficient of the bias term always contains λ we cannot make the regularization zero

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 θ^*_λ that minimizes $J(\theta)$ over a range of λ. 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 λ. Your goal is to find λ that gives the minimum test loss. It's hard to predict what λ 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 λ.

   Solution:
   As we can see in figure 6 the test loss is minimum for lambda = 10^{-2}.

8. What θ would you select for deployment and why?

   We would choose the θ which produced the minimum test loss when $\lambda = 10^{-2}$ for deployment as it has produced the best results. $\theta =$

```
[-1.15463918  0.49763931  1.35129771  2.1852422  -1.6482735  -0.77561703
 -0.77398786 -0.77398786  0.68333382  1.33888896  2.25458266 -0.39821028
 -1.33593441 -3.67468185  1.39773213  2.23016977  1.24397633  0.46450449
 -0.0513538  -0.0513538  -0.0513538  -0.02349766 -0.02349766 -0.02349766
  0.00773689  0.00773689  0.00773689  0.02295154  0.02295154  0.02295154
  0.03162864  0.03162864  0.03162864 -0.03442828 -0.03442828 -0.03442828
  0.09361602  0.09361602  0.09361602  0.07643784  0.07643784  0.07643784
  0.06866328  0.06866328  0.06866328  0.06440173  0.06440173  0.06440173
 -1.22092363]
```

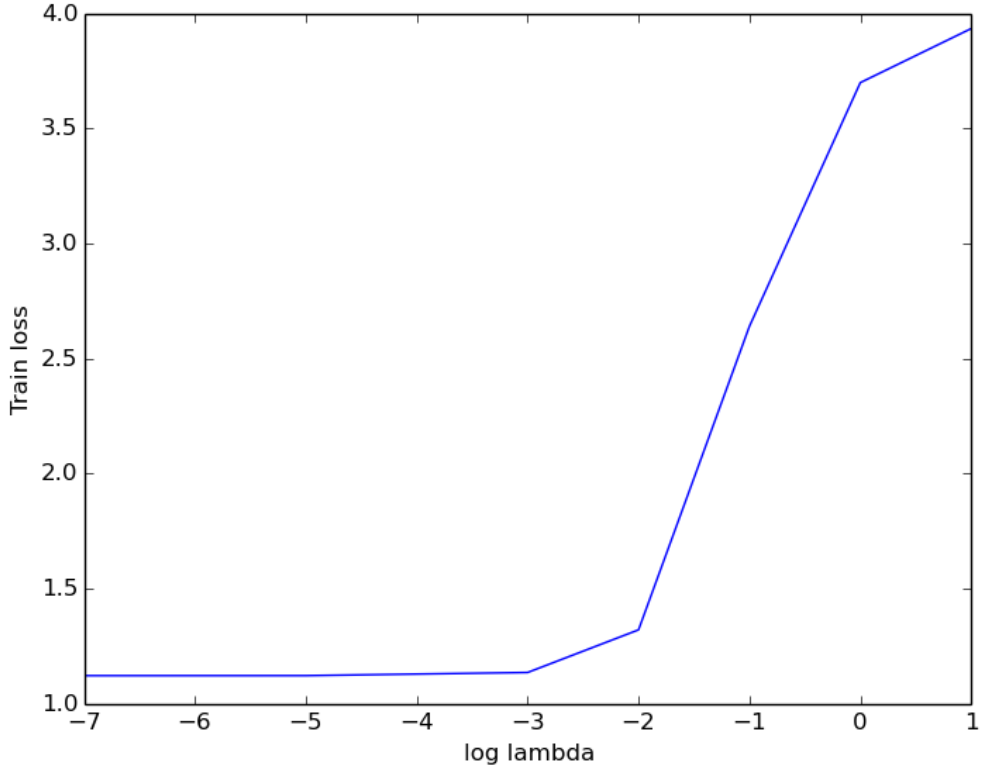

Figure 5: Plot of training loss v/s $log_{10}\lambda$

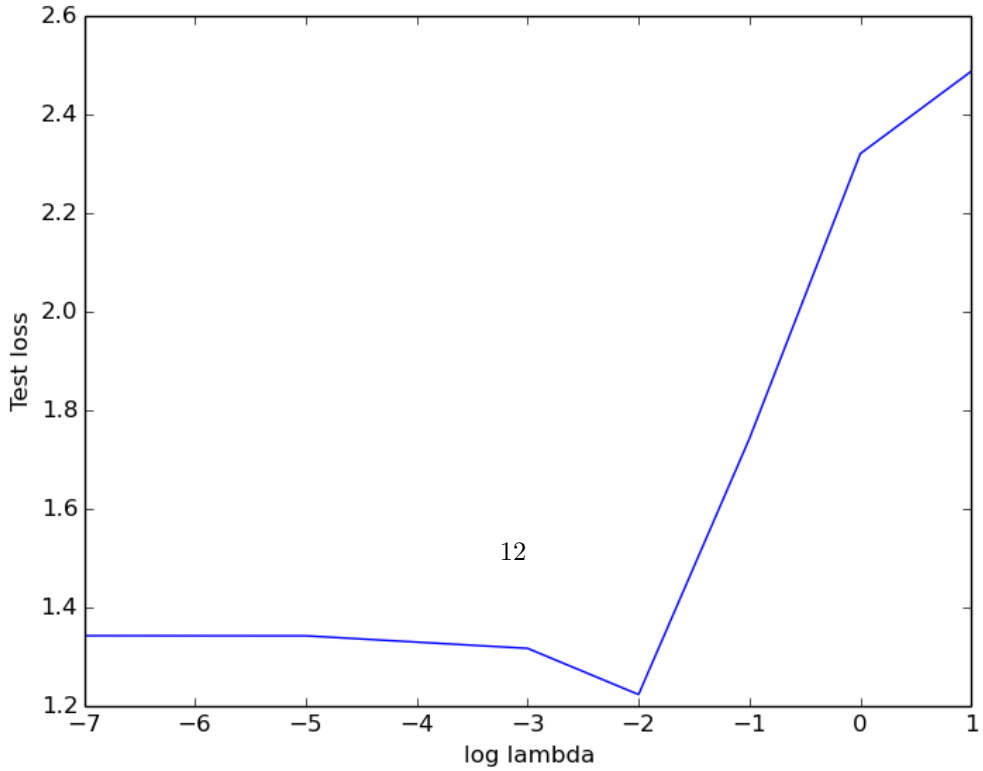

Figure 6: Plot of test loss v/s $log_{10}\lambda$

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. *W*rite down the update rule for θ in SGD for the ridge regression objective function.

$$\theta_{i+1} = \theta_i - \eta \left[X^T (X\theta - y) + 2\lambda\theta \right]$$

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 stochastic_grad_descent(X, y, alpha=0.05, 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 the
    loss_hist = np.zeros((num_iter, num_instances)) #Initialize loss_hist

    batch_size=1
    sample_size=100

    for i in range(num_iter):

        for j in range(int(sample_size/batch_size)):

            x_batch=X[j]
            y_batch=y[j]

            grad = compute_regularized_square_loss_gradient(x_batch, y_batch, theta

            theta = theta - (alpha * grad)

            theta_hist[i][j] = theta
            loss_hist[i][j] = compute_square_loss(x_batch, y_batch, theta)
            loss_hist[i][j] = loss_hist[i][j] + lambda_reg * np.dot(theta.T, theta)

    return(loss_hist, theta_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. (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.

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

   It takes about 10^-3 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?

   Solution:
   If we want to minimize the total number of epochs we would choose SGD as it reaches a value near to the minimum in less number of epochs. However, if we want to minimize the time we would choose gradient descent as it reaches the minimum value faster than SGD as every step in case of Gradient Descent takes less time.

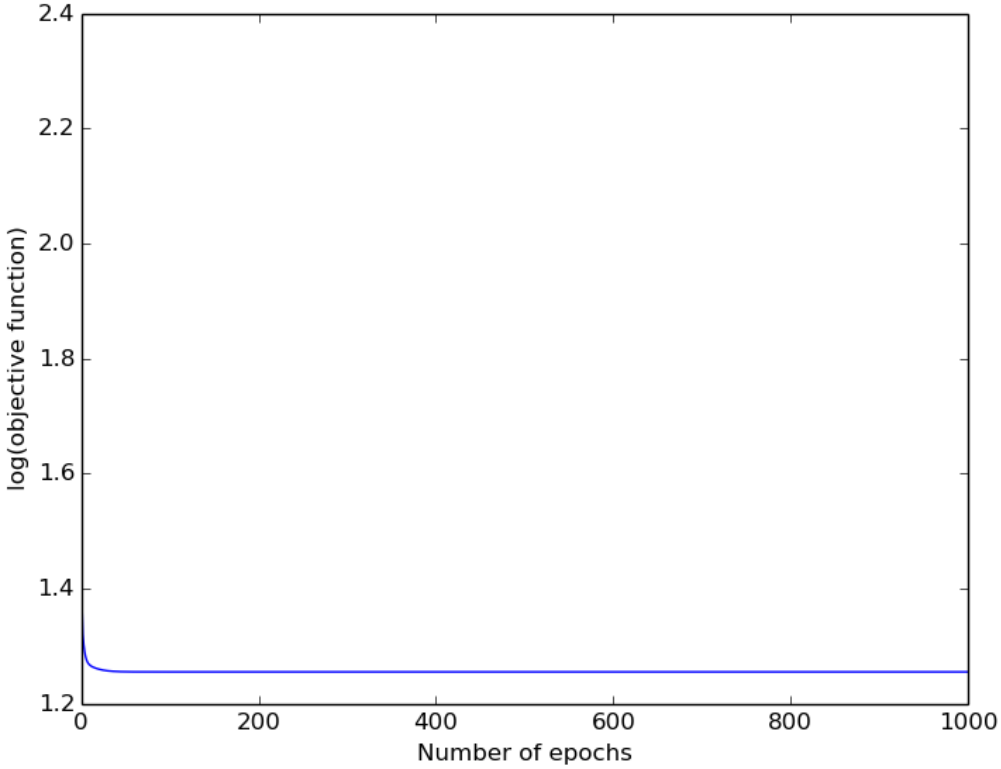

Figure 7: log(objective function) v/s epochs with $\eta = 0.05$

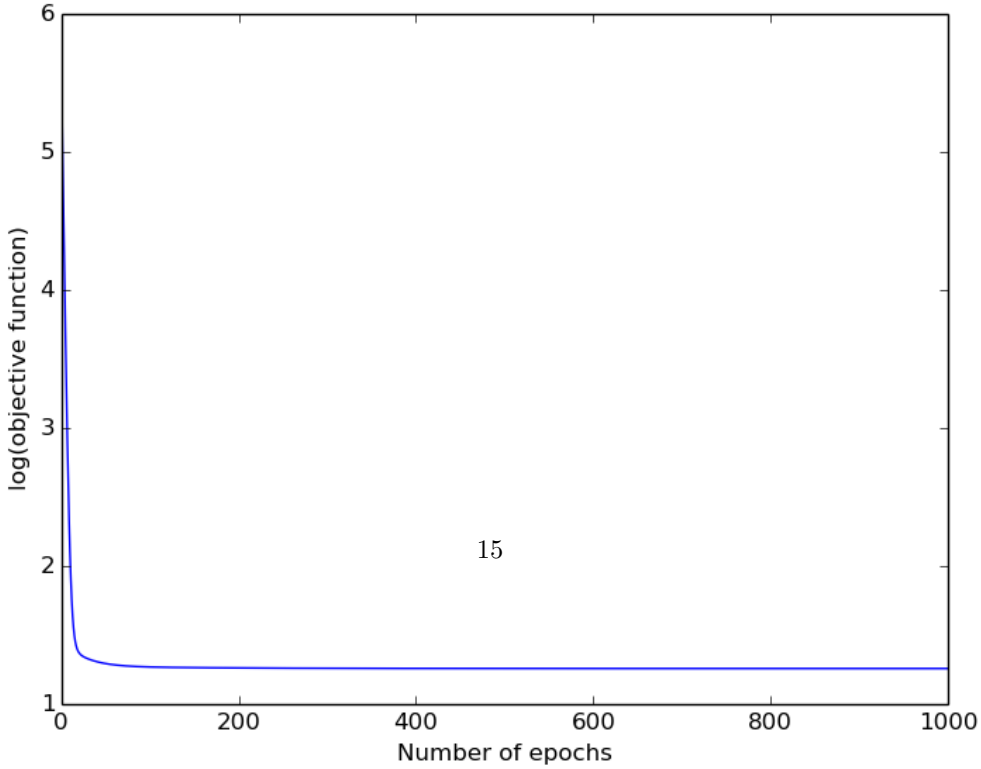

Figure 8: log(objective function) v/s epochs with $\eta = 0.005$

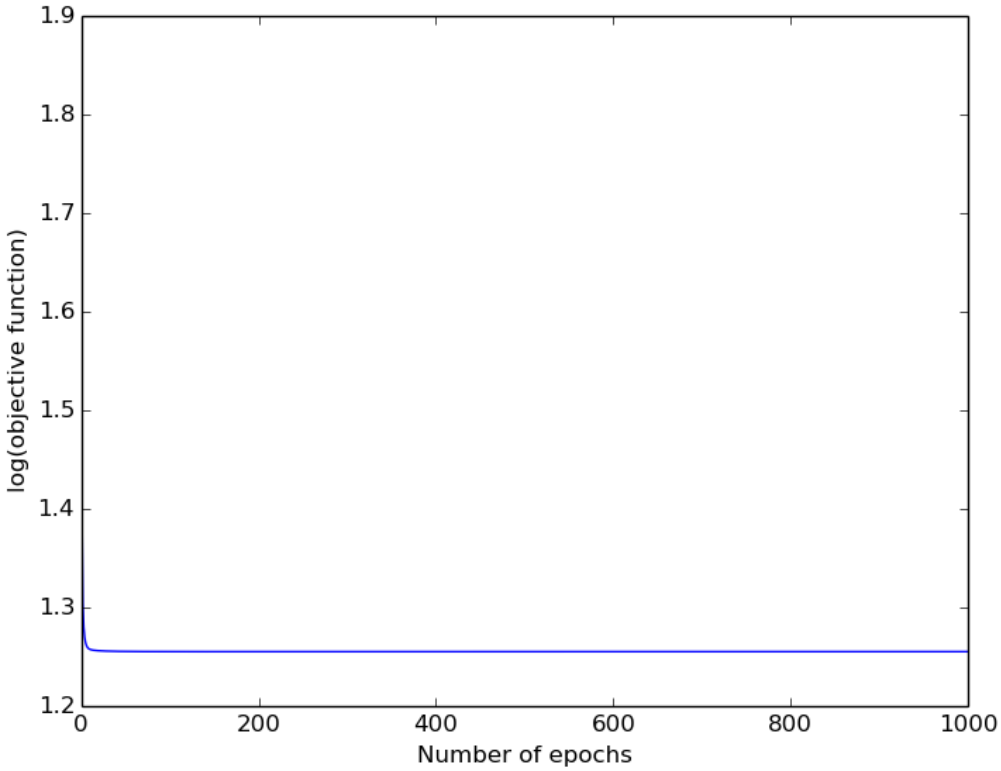

Figure 9: log(objective function) v/s epochs with $\eta = 1/t$

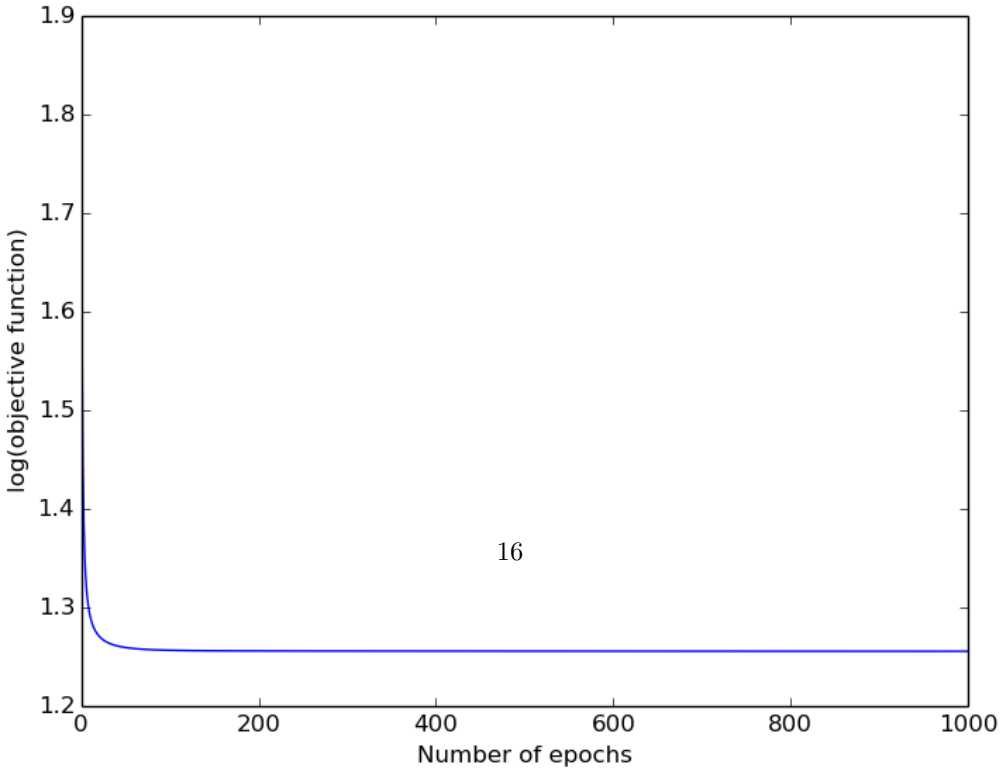

Figure 10: log(objective function) v/s epochs with $\eta = 1/\text{sqrt}(t)$

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

$$h(a) = \mathbb{E}\,(a - y)^2$$
$$h(a) = \mathbb{E}\left(a^2 + y^2 - 2ay\right)$$
$$h(a) = \mathbb{E}\left(a^2\right) + \mathbb{E}\left(y^2\right) - \mathbb{E}\,(2ay)$$
$$h(a) = a^2 + \mathbb{E}\left(y^2\right) - 2a\mathbb{E}\,(y)$$

Equating h'(a) = 0, will give us the optimum value of a for which the risk in minimum.

$$h'(a) = 2a - 2\mathbb{E}\,(y) = 0$$
$$a = \mathbb{E}\,(y)$$

Substituting this value back in the original loss function, we can get the minimum risk

$$h(a) = [\mathbb{E}\,(y)]^2 + \mathbb{E}\left(y^2\right) - 2\,[\mathbb{E}\,(y)]^2$$
$$h(a) = \mathbb{E}\left(y^2\right) - [\mathbb{E}\,(y)]^2$$
$$h(a) = var(y)$$

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

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

where $(x, y) \sim P_{\mathcal{X}\times\mathcal{Y}}$, and the **Bayes decision function** $f^* : \mathcal{X} \to \mathcal{A}$ 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 $\mathcal{A} = \mathcal{Y} = \mathbf{R}$. 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 $f^*(x) = \arg\min_a \mathbb{E}\left[(a-y)^2 \mid x\right]$, where the expectation is with respect to y.

$$\arg\min_a \mathbb{E}\left[(a-y)^2 \mid x\right] = \mathbb{E}\left[(a - \mathbb{E}[y \mid x] + \mathbb{E}[y \mid x] - y)^2 \mid x\right]$$
$$= \mathbb{E}[a - \mathbb{E}[y \mid x] \mid x]^2 + \mathbb{E}[\mathbb{E}[y \mid x] - y \mid x]^2 + 2\mathbb{E}[a - \mathbb{E}[y \mid x] \mid x]\,\mathbb{E}[\mathbb{E}[y \mid x] - y \mid x]$$

Now the last term becomes 0, so we get

$$\arg\min_a \mathbb{E}\left[(a-y)^2 \mid x\right] = \mathbb{E}[a - \mathbb{E}[y \mid x] \mid x]^2 + \mathbb{E}[\mathbb{E}[y \mid x] - y \mid x]^2$$

Now, for this expression to be minimum, we have to put a $= \mathbb{E}[y \mid x]$

$$f^*(x) = \mathbb{E}[y \mid x]$$

(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

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

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.

In the last part we proved that,

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

for all x and every f

If we take expectation on both the sides we get,

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

which means that $f^*(x)$ is the best possible function which gives minimum loss.

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.