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

Due: Friday, February 5, 2015, at 6pm (Submit via NYU Classes)

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. You may include your code inline or submit it as a separate file. You may either scan hand-written work or, preferably, write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython).

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: `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`
- 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)
- Investigate what happens to the convergence rate when you intentionally make the feature values have vastly different orders of magnitude. Try a dataset (could be artificial) where $\mathcal{X} \subset \mathbf{R}^2$ so that you can plot the convergence path of GD and SGD. Take a look at `http://imgur.com/a/Hqolp` for inspiration.
- 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"?

Include any investigations you do in your submission, and we may award optional credit.

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."

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, 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 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 featurenormalization(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:
        trainnormalized - training set after normalization
        testnormalized - 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]

    # Normalize (uses array broadcasting http://wiki.scipy.org/
        EricsBroadcastingDoc)
    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 $h_\theta : \mathbf{R}^d \to \mathbf{R}$, 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.
   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 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.]
   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 θ in the gradient descent algorithm. Let η 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 θ.
   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. 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:Skip

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

8. Using your small dataset, verify that your `compute_square_loss_gradient` function returns the correct value.
Solution:Skip

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:

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

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. Running the gradient checker takes extra time. In practice, once you're convinced your gradient calculator is correct, you can stop calling the checker.
   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
    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)
```

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

```
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_checking:
            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[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.
   Solution:

   alpha = 0.5 diverges.

1. (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

[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.

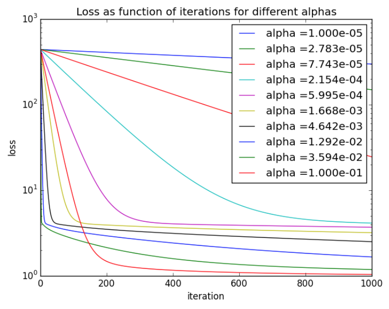

Figure 1:

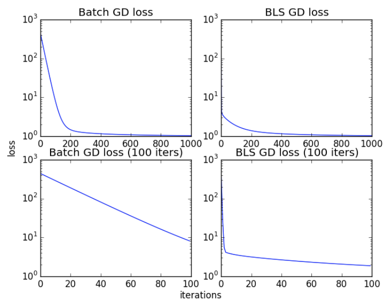

Figure 2:
BLS = Backtracking Line Search

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.)
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
```

Backtrack line search converges in less iterations.

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

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

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

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

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

   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):
    #TODO
    num_instances = y.shape[0]
    grad = 1.0/num_instances * np.dot((np.dot(X, theta.T) - y), X)
        + 2*lambda_reg*t
    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)+np.sum(theta
        **2)*lambda_reg/(2*num_instances)
    return theta_hist, loss_hist
```

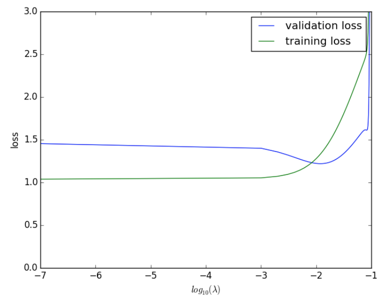

Figure 3:

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

   Solution:

   If we increase B, the corresponding coefficient β_0 will decrease to result in the same optimal bias. With a smaller coefficient, there is less effect of regularization on the bias term. As $B \to \infty$, $\beta_0 \to 0$ and the regularization effect approaches 0.

5. Now fix $B = 1$. Choosing a reasonable step-size or using backtracking line search, find the θ^*_λ that minimizes $J(\theta)$ for a range of λ. You should plot the training loss and the validation loss (just the square loss part, without the regularization) as a function of λ. Your goal is to find λ that gives the minimum validation 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, $\lambda \in \left\{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\right\}$. 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:

   The optimal λ is around 10^{-2}.

6. (Optional) Once you have found a good value for λ, repeat the fits with different values for B, and plot the results. For this dataset, does regularizing the bias help, hurt, or make no significant difference?

7. (Optional) Estimate the average time it takes on your computer to compute a single gradient step.

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

   Solution:

   Choose the θ that minimizes the validation 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 we cycling through the training examples, it is important to shuffle them 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.

   Solution:

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

2. Implement `stochastic_grad_descent`.
   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):
```

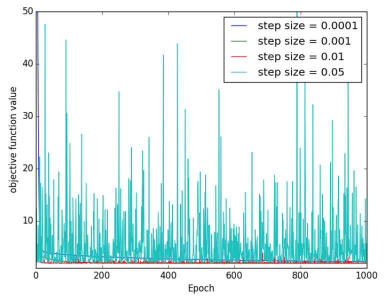

Figure 4:

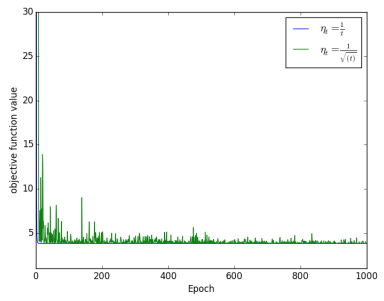

Figure 5:

```
        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
```

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? (Note: In this case we are investigating the convergence rate of the optimization algorithm, thus we're interested in the value of the objective function, which includes the regularization term.)

4. (Optional) Try a stepsize rule of the form $\eta_t = \frac{\eta_0}{1+\eta_0 \lambda t}$, where λ is your regularization constant,

and η_0 a constant you can choose. How do the results compare?

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

   Solution:

   About 10^{-3} seconds

6. 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:

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

3 Risk Minimization

Recall that the definition of 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 $\mathcal{A} = \mathcal{Y} = \mathbf{R}$.

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

$$\begin{aligned}
\mathbb{E}\left[(\hat{y} - y)^2 \mid x\right] &= \mathbb{E}\left[(\hat{y} - m + m - y)^2 \mid x\right] \\
&= \mathbb{E}\left[(\hat{y} - m)^2 + 2(\hat{y} - m)(m - y) + (m - y)^2 \mid x\right] \\
&= \mathbb{E}\left[(\hat{y} - m)^2 \mid x\right] + 2\mathbb{E}\left[(\hat{y} - m)(m - y) \mid x\right] + \mathbb{E}\left[(m - y)^2 \mid x\right].
\end{aligned}$$

   After conditioning on x, neither $\hat{y}$ nor m is random. So the middle term is

$$\begin{aligned}
\mathbb{E}\left[(\hat{y} - m)(m - y) \mid x\right] &= (\hat{y} - m)\underbrace{(m - \mathbb{E}[y \mid x])}_{=0} \\
&= 0
\end{aligned}$$

   So

$$\mathbb{E}\left[(\hat{y} - y)^2 \mid x\right] = \mathbb{E}\left[(\hat{y} - m)^2 \mid x\right] + \mathbb{E}\left[(m - y)^2 \mid x\right].$$

This expression is minimized by taking $\hat{y} = m$. Thus for any x, the expected loss conditional on x is

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

Since this is the minimizer for each x, it is certainly the minimizer in expectation over x. Written out mathematically, we are saying the following: We have shown that

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

for every x and for every f. Then taking expectations on each side, we get

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

which implies that f^* is the risk minimizer.

2. (Optional) Show that for the absolute loss $\ell(\hat{y}, y) = |y - \hat{y}|$, the Bayes decision function is a $f^*(x) = \text{median}\left[Y \mid X = x\right]$. [Hint: Again, consider one x at time. For some approaches, it may help to use the following characterization of a median: m is a median of the distribution for random variable Y if $P(Y \ge m) \ge \frac{1}{2}$ and $P(Y \le m) \ge \frac{1}{2}$.] Note: This loss function leads to "median regression", There are other loss functions that lead to "quantile regression" for any chosen quantile.
Solution:
Let $m = \text{median}\left(Y \mid X = x\right)$. 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 $Y \le m$, then

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

If $Y > m$, then

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

Putting this together, we get:

$$\begin{aligned} |Y - b| - |Y - m| &\ge (b - m)\,1(Y \le m) - (b - m)\,1(Y > m) \\ &= (b - m)\left[1(Y \le m) - 1(Y > m)\right] \end{aligned}$$

Taking expectations yields

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

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:

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

Taking expectations yields

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

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