Machine Learning and Computational Statistics
Homework 1: Mathematical Fundamentals, Ridge Regression, Gradient Descent, and SGD

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. If you are using LyX, then the listings package tends to work better.

1 Introduction

In this homework you will first solve some probability and linear algebra questions and then 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"?

2 Mathematical Fundamentals

The following questions are designed to check how prepared you are to take this class. Familiarity with linear algebra and probability at the level of these questions is expected for the class.

2.1 Probability

Let $(X_1, X_2, \cdots, X_d)$ have a d-dimensional multivariate Gaussian distribution, with mean vector $\mu \in \mathbf{R}^d$ and covariance matrix $\Sigma \in \mathbf{R}^{d \times d}$, i.e. $(X_1, X_2, \cdots, X_d) \sim \mathcal{N}(\mu, \Sigma)$ Use μ_i to denote the ith element of μ and Σ_{ij} to denote the element at the ith row and j^{th} column of Σ.

1. Let $x, y \in \mathbf{R}^d$ be two independent samples drawn from $\mathcal{N}(\mu, \Sigma)$. Give expression for $\mathbb{E}\|x\|_2^2$ and $\mathbb{E}\|x - y\|_2^2$. Express your answer as a function of μ and Σ.

   Solution:

$$\mathbb{E}\|x\|_2^2 = \sum_{i=1}^d \mathbb{E}(x_i^2)$$
$$= \sum_{i=1}^d var(x_i) + (\mathbb{E}x_i)^2$$
$$= \sum_{i=1}^d (\mu_i^2 + \Sigma_{ii})$$
$$\mathbb{E}\|x - y\|_2^2 = \mathbb{E}[(x - y)^T (x - y)]$$
$$= \mathbb{E}[x^T x + y^T x - 2x^T y]$$
$$= \mathbb{E}\|x\|_2^2 + \mathbb{E}\|y\|_2^2 - 2\mu^T \mu$$
$$= 2 \sum_{i=1}^d \Sigma_{ii}$$

   Note that $\mathbb{E}\|x - y\|_2^2$ does not depend on μ. Can you given a non-mathematical explanation?

2. Find the distribution of $Z = \alpha_i X_i + \alpha_j X_j$, for $i \neq j$ and $1 \leq i, j \leq d$. The answer will belong to a familiar class of distribution. Report the answer by identifying this class of distribution and specifying the parameters.

   Solution: We use the fact that if some real valued random variables are jointly Gaussian, then their linear combination is Gaussian. (Interestingly, the converse is also true - if all linear combination of real valued random variables are gaussian, their joint is also gaussian).

   The mean of Z is $(\alpha_i \mu_i + \alpha_j \mu_j)$ and the variance of Z is $(\alpha_i^2 \Sigma_{ii} + \alpha_j^2 \Sigma_{jj} + 2\alpha_i \alpha_j \Sigma_{i,j})$.

3. (Optional) Assume W and R are two Gaussian distributed random variables. Is $W + R$ still Gaussian? Justify your answer.

   Solution: Not necessary. Suppose W is a Gaussian distributed random variable and $\epsilon = \pm 1$ with probability $1/2$. Let $R = \epsilon W$. Then R is also a Gaussian distributed random variable. But $W + R$ is equal to zero with probability $1/2$ and 2W with probability $1/2$.

2.2 Linear Algebra

1. Let A be a $d \times d$ matrix with rank k. Consider the set $S_A := \{x \in \mathbf{R}^d | Ax = 0\}$. What is the dimension of S_A?

Solution:

The dimension of S_A is $(d - k)$. (Rank + Nullity theorem)

2. Assume S_v is a k dimensional subspace in $\mathbf{R}^d$ and $v_1, v_2, \cdots, v_k$ form an orthonormal basis of S_v. Let w be an arbitrary vector in $\mathbf{R}^d$. Find

$$x^* = \operatorname*{argmin}_{x \in S_v} \|w - x\|,$$

where $\|w - x\|$ is the Euclidean distance between w and x. Express x^* as a function of $v_1, v_2, \ldots, v_n$ and w.

Solution: Since $x^* \in S_v$) there exists unique (why?) $\alpha_1, \alpha_2, \ldots, \alpha_k$ such that $x^* = \sum_{i=1}^k \alpha_i v_i$.

$$
\begin{aligned}
\|w - x\|_2^2 &= (w - x)^T (w - x) \\
&= \|w\|_2^2 + \|x\|_2^2 - 2w^T x \\
&= \|w\|_2^2 + \sum_{i=1}^k \alpha_i^2 - 2 \sum_{i=1}^k \alpha_i w^T v_i
\end{aligned}
$$

$$\frac{\partial}{\partial \alpha_i} \|w - x\|_2^2 = 2\alpha_i - 2w^T v_i = 0$$

$$x^* = \sum_{i=1}^k (v_i \cdot x) v_i$$

Or can can just identify this as the projection of w to the $span(v_1, v_2, \ldots, v_k)$

3. (Optional) Continuing from above, x^* can be expressed as

$$x^* = Mw,$$

where M is a $d \times d$ matrix. Prove that such an M always exists or more precisely find an expression for M as a function of $v_1, v_2, \cdots, v_k$. Compute the eigenvalues and one set of eigenvectors of M corresponding to the nonzero eigenvalues.

Solution:

$$M = \sum_{i=1}^k v_i v_i^T.$$

M has k eigenvalues that equal to 1 and $(d - k)$ eigenvalues that equal to 0. One set of eigenvectors is $\{v_1, v_2, \cdots, v_k\}$.

3

3 Linear Regression

3.1 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 are treated as "more important", which is not usually what you want. One common approach to feature normalization is perform an affine transformation (i.e. shift and rescale) on 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?) Note that a feature with constant value cannot be normalized in this way. Your function should discard features that are constant in the training set.

Solution:

```python
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 (num_instances,
    num_features)
        test - test set, a 2D numpy array of size (num_instances,
    num_features)
    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]
```

---

[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 we 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/
    EricsBroadcastingDoc )
    train_normalized = (train − train_min) / (train_max − train_min)
    test_normalized = (test − train_min) / (train_max − train_min)

    return train_normalized, test_normalized
```

3.2   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 "average square loss" objective function:

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

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 the "response". Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign. [Being able to write expressions as matrix/vector expressions without summations is crucial to making implementations that are useful in practice, since you can use numpy (or more generally, an efficient numerical linear algebra library) to implement these matrix/vector operations orders of magnitude faster than naively implementing with loops in Python.]

    Solution:

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

2. Write down an expression for the gradient of J (again, as a matrix/vector expression, without using an explicit summation sign).

    Solution:

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

3. In our search for a θ that minimizes J, suppose we take a step from θ to $\theta + \eta h$, where $h \in \mathbf{R}^{d+1}$ is the "step direction" (recall, this is not necessarily a unit vector) and $\eta \in (0, \infty)$ is the "step size" (note that this is not the actual length of the step, which is $\eta\|h\|$). Use the

gradient to write down an approximate expression for the change in objective function value $J(\theta + \eta h) - J(\theta)$. [This approximation is called a "linear" or "first-order" approximation.]

Solution:

$$J(\theta + \eta h) - J(\theta) \approx \nabla J(\theta)^T (\theta + \eta h - \theta) = \eta \nabla J(\theta)^T h$$

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

   Solution:

   $$\theta_{n+1} = \theta_n - \eta \nabla J(\theta_n) \text{ , where } \eta > 0$$

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.

   Solution:

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

```python
def compute_square_loss_gradient(X, y, theta):
    """
    Compute the gradient of the average 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 numpy array of size (
    num_features)
    Returns:
        grad − gradient vector, 1D numpy array of size (
    num_features)
    """
    #TODO
    num_instances = y.shape[0]
    grad = 2.0 / num_instances * np.dot((np.dot(X, theta) − y), X)
    return grad
```

3.3   (OPTIONAL) 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 vector $h \in \mathbf{R}^d$, the directional derivative of J at θ in the direction h is given by[2]

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

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 $h = (1, 0, 0, \ldots, 0)$ to get the first component of the gradient. Then take $h = (0, 1, 0, \ldots, 0)$ 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.

Solution:

```
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e−4):
    """Implement Gradient Checker
    Check that the function compute_square_loss_gradient returns
    the correct gradient for the given X, y, and theta.
```

---

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

```python
    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 indicating 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)

    error = np.linalg.norm(true_gradient - approx_grad)
    return (error < tolerance)
```

```python
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
    #TODO
    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)
```

```
        error = np.linalg.norm(true_gradient - approx_grad)
        return (error < tolerance)
```

3.4 Batch Gradient Descent[3]

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_step=1000, grad_check=
    False):
    """
    In this question you will implement batch gradient descent to
    minimize the average 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
        num_step - number of steps to run
        grad_check - a boolean value indicating whether checking
    the gradient when updating
    Returns:
        theta_hist - the history of parameter vector, 2D numpy
    array of size (num_step+1, num_features)
                     for instance, theta in step 0 should be
    theta_hist[0], theta in step (num_step) is theta_hist[-1]
        loss_hist - the history of average square loss on the data,
     1D numpy array, (num_step+1)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize
    theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    theta = np.zeros(num_features) #Initialize theta

    #TODO
    for i in range(num_step):
        theta_hist[i] = theta
```

[3]Sometimes people say "batch gradient descent" or "full batch gradient descent" to mean gradient descent, defined as we discussed in class. They do this to distinguish it from stochastic gradient descent and minibatch gradient descent, which they probably use as their default.

```
        loss_hist[i] = compute_square_loss(X, y, theta)
        grad = compute_square_loss_gradient(X, y, theta)   #Compute
the gradient

        if grad_check:
            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 #Update theta
        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[4]. 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 average square loss as a function of the number of steps for each step size. Briefly summarize your findings.

Solution:

The gradient descent diverges when $\alpha = 0.5$ or $\alpha = 0.1$. When converging, the one with a larger step size converges faster.

---

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

Average square loss as a function of steps in BGD
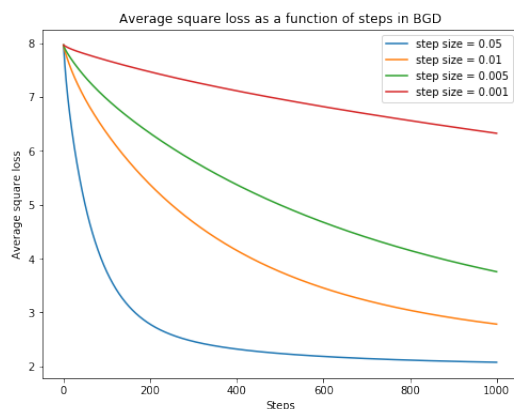
3. (Optional) Implement backtracking line search (google it). How does it compare to the best fixed step-size you found in terms of number of steps? In terms of time? How does the extra time to run backtracking line search at each step compare to the time it takes to compute the gradient? (You can also compare the operation counts.)

Solution:

Backtrack line search converges in less steps. It yields a slightly lower, though not significantly, average square loss after 1000 steps.

```
#TODO
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
```

11

Average square loss as a function of steps in BGD
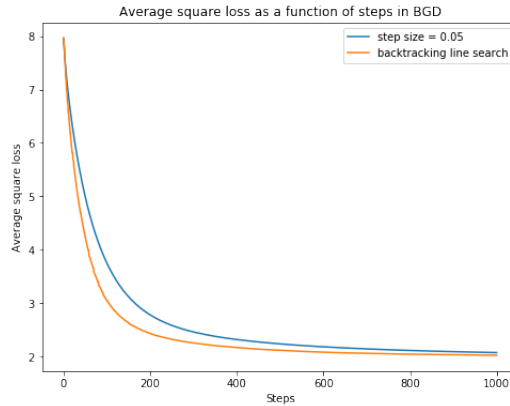
3.5 Ridge Regression (i.e. Linear Regression with ℓ_2 regularization)

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

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

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. (Matrix/vector expression – no summations please.)

   Solution:

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

2. Implement `compute_regularized_square_loss_gradient`.

   Solution:

```
def compute_regularized_square_loss_gradient(X, y,
    lambda_reg):
    """
     Compute the gradient of L2-regularized average square loss
    function given X, y and 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 numpy array of size (
    num_features)
        lambda_reg - the regularization coefficient
    Returns:
        grad - gradient vector, 1D numpy array of size (
    num_features)
    """
    #TODO
    num_instances = y.shape[0]
    grad = 2.0 / num_instances * np.dot((np.dot(X, theta) - y), X)
    + 2 * lambda_reg * theta
    return grad
```

3. Implement `regularized_grad_descent`.

Solution:

```
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2,
    num_step=1000):
    """
    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
        lambda_reg - the regularization coefficient
        num_step - number of steps to run
    Returns:
        theta_hist - the history of parameter vector, 2D numpy
    array of size (num_step+1, num_features)
                    for instance, theta in step 0 should be
    theta_hist[0], theta in step (num_step+1) is theta_hist[-1]
        loss hist - the history of average square loss function
    without the regularization term, 1D numpy array.
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.zeros(num_features) #Initialize theta
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize
    theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    #TODO
    for i in range(num_step):
        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 θ_0 will have to decrease by the same factor to end up with the same function f. A smaller coefficient θ_0 incurs less regularization penalty. As $B \to \infty$, $\beta_0 \to 0$ 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 $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \mathbf{R}$, loss function $\ell : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ and hyperparameters $B, \lambda_1, \lambda_2 > 0$, let

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

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 $h_B(x) = \theta_{1,B}^* B + \left(\theta_{2,B}^*\right)^T x$.

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 $\frac{1}{B^2}$:

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

The corresponding prediction function is $h_1(x) = \theta_{1,1}^* + \left(\theta_{2,1}^*\right)^T x$. 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 θ_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 θ_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

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

where $\lambda_1, \lambda_2 > 0$.

So we want to show that if

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

and

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

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

Note that

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

So since

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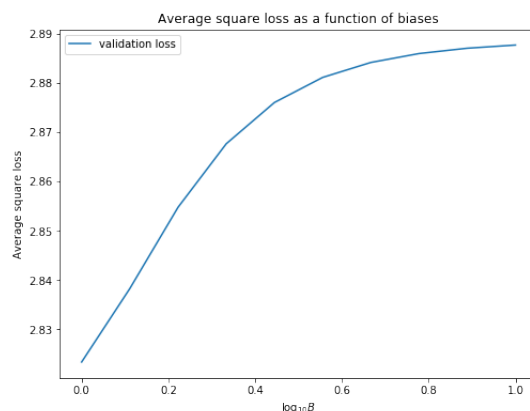\end{aligned}
$$

and

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

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.
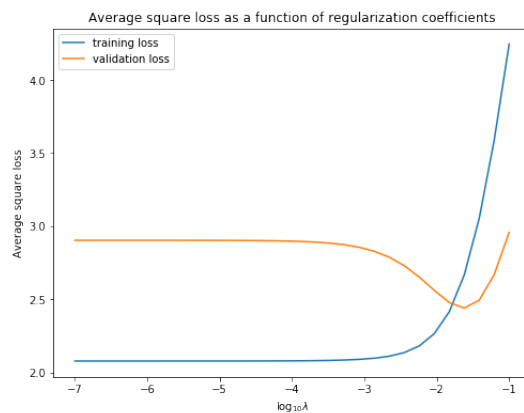
   Solution:

   The optimal B is 1.

Average square loss as a function of biases

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 average square loss and the test average square loss (just the average square loss part, without the regularization, in each case) as a function of λ. Your goal is to find λ that gives the minimum average square loss on the test set. 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. [If you like, you may use sklearn to help with the hyperparameter search.]

Solution:

The optimal λ is around $10^{-1.6}$.

Average square loss as a function of regularization coefficients

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

Solution:

Choose the θ that minimizes the average square loss on the test set. It provides a better generalization, since we have never trained the data on the test set.

3.6 Stochastic Gradient Descent

When the training data set is very large, evaluating the gradient of the objective function can take a long time, since it requires looking at each training example to take a single gradient step. When the objective function takes the form of an average of many values, such as

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} f_i(\theta)$$

(as it does in the empirical risk), stochastic gradient descent (SGD) can be very effective. In SGD, rather than taking $-\nabla J(\theta)$ as our step direction, we take $-\nabla f_i(\theta)$ for some i chosen uniformly at random from $\{1, \ldots, m\}$. The approximation is poor, but we will show it is unbiased.

In machine learning applications, each $f_i(\theta)$ would be the loss on the ith example (and of course we'd typically write n instead of m, for the number of training points). In practical implementations for ML, the data points are **randomly shuffled**, and then we sweep through the whole training set one by one, and perform 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. You can use the same ordering for each epoch, though optionally you could investigate whether reshuffling after each epoch affects the convergence speed.

1. Show that the objective function

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

can be written in the form $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} f_i(\theta)$ by giving an expression for $f_i(\theta)$ that makes the two expressions equivalent.

Solution:

$$
\begin{aligned}
J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x_i) - y_i \right)^2 + \lambda \theta^T \theta \\
&= \frac{1}{m} \sum_{i=1}^{m} \left[\left(h_\theta(x_i) - y_i \right)^2 + \lambda \theta^T \theta \right]
\end{aligned}
$$

so

$$f_i(\theta) = \left(h_\theta(x_i) - y_i \right)^2 + \lambda \theta^T \theta.$$

2. Show that the stochastic gradient $\nabla f_i(\theta)$, for i chosen uniformly at random from $\{1, \ldots, m\}$, is an **unbiased estimator** of $\nabla J(\theta)$. In other words, show that $\mathbb{E}\left[\nabla f_i(\theta) \right] = \nabla J(\theta)$ for any

17

θ. (Hint: It will be easier, notationally, to prove this for a general $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta)$, rather than the specific case of ridge regression. You can start by writing down an expression for $\mathbb{E}\left[\nabla f_i(\theta)\right]$...)

Solution:

$$
\begin{aligned}
\mathbb{E}\left[\nabla f_i(\theta)\right] &= \sum_{j=1}^{m} \mathbb{P}\left(i = j\right) \nabla f_i(\theta) \\
&= \frac{1}{m}\sum_{j=1}^{m} \nabla f_i(\theta) \\
&= \nabla J(\theta)
\end{aligned}
$$

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

   Solution:

   $$\nabla f_i(\theta) = 2\left(h_\theta(x_i) - y_i\right)x_i + 2\lambda\theta$$

   So

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

4. Implement `stochastic_grad_descent`. (Note: You could potentially generalize the code you wrote for batch gradient to handle minibatches of any size, including 1, but this is not necessary.)

   Solution:

```python
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2,
    num_epoch=1000, eta0=False):
    """
    In this question you will implement stochastic gradient descent
     with regularization term
    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 - string or float, step size in gradient descent
                NOTE: In SGD, it's not a good idea to use a fixed
    step size. Usually it's set to 1/sqrt(t) or 1/t
                if alpha is a float, then the step size in every
    step is the float.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t).
                if alpha == "1/t", alpha = 1/t.
        lambda_reg - the regularization coefficient
        num_epoch - number of epochs to go through the whole
    training set
    Returns:
```

18

```python
    theta_hist - the history of parameter vector, 3D numpy
array of size (num_epoch, num_instances, num_features)
                    for instance, theta in epoch 0 should be
theta_hist[0], theta in epoch (num_epoch) is theta_hist[-1]
    loss_hist - the history of loss function vector, 2D numpy
array of size (num_epoch, num_instances)
"""
num_instances, num_features = X.shape[0], X.shape[1]
theta = np.ones(num_features) #Initialize theta

theta_hist = np.zeros((num_epoch, num_instances, num_features))
#Initialize theta_hist
loss_hist = np.zeros((num_epoch, num_instances)) #Initialize
loss_hist
#TODO
# Set step counter
step_size_p = 1

np.random.seed(10)
for i in range(num_epoch):
    # Shuffle points
    shuffle = np.random.permutation(num_instances)

    for j in shuffle:
        # Store the historical theta
        theta_hist[i, j] = theta
        loss_hist[i, j] = compute_square_loss(X, y, theta) + np
.sum(theta**2)*lambda_reg

        # Simultaneously update theta
        grad = 2*(np.dot(X[j], theta.T) - y[j])*X[j] + 2*
lambda_reg*theta #Compute gradient of one single point

        if eta0:
            theta = theta - eta0 / (1+eta0*lambda_reg*
step_size_p) * grad #Update theta
            step_size_p += 1
        else:
            if isinstance(alpha, str):
                if alpha == "1/t":
                    theta = theta - 0.1/step_size_p * grad #
Update theta
                elif alpha == "1/sqrt(t)":
                    theta = theta - 0.1/np.sqrt(step_size_p) *
grad #Update theta
                    step_size_p+=1
```

```
            else:
                theta = theta - alpha * grad #Update theta

    return theta_hist, loss_hist
```

5. 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: $\eta_t = \frac{C}{t}$ and $\eta_t = \frac{C}{\sqrt{t}}$, $C \leq 1$. Please include $C = 0.1$ in your submissions. You're encouraged to try different values of C (see notes below for details). 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, if you prefer) for each of the approaches to step size. How do the results compare?

Some things to note:

- 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.

- Sometimes the initial step size (C for C/t and $C/\sqrt{t}$) is too aggressive and will get you into a part of parameter space from which you can't recover. Try reducing C to counter this problem.

- 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 terminology we'll learn in Lecture 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). 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 minimizer.

- (Optional) There is another variant of SGD, sometimes called **averaged SGD**, in which rather than using the last parameter value we visit, say θ^T, we use the average of all parameter values we visit along the optimization path: $\theta = \frac{1}{T} \sum_{t=1}^T \theta^t$, where T is total number of steps taken. Try this approach[5] and see how it compares.

Solution:

The step size of $\frac{0.1}{\sqrt{t}}$ performs the best. The step size of $\frac{0.1}{t}$ does not converge. For both fixed step sizes of 0.01 and 0.001, SGD loss fluctuates, though with smaller scale for 0.001.

---

[5]Some theory for averaged SGD is given on page 191 of Understanding Machine Learning: From Theory to Algorithms. Refer to page 195 of the same book for other averaging techniques you can try.

log(ridge objective function) as a function of epochs in SGD
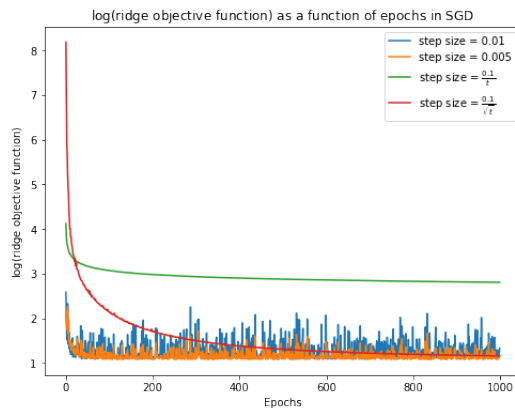
6. (Optional) Try a stepsize rule of the form $\eta_t = \frac{\eta_0}{1+\eta_0\lambda t}$, where λ is your regularization constant, and η_0 a constant you can choose. How do the results compare?

Solution:

The stepsize rule with $\eta_0 = 0.01$ performs better than the best step size chosen in the previous question, having a faster convergence.

log(ridge objective function) as a function of epochs in SGD
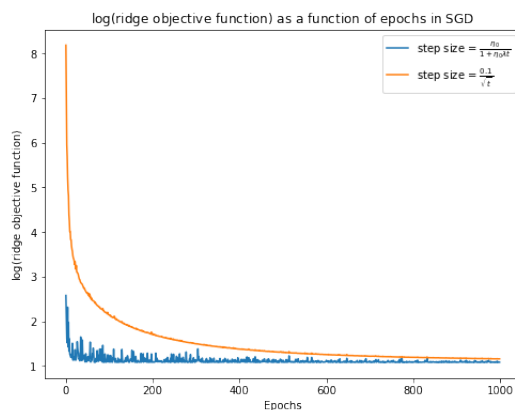
4 Risk Minimization

4.1 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

21

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, the best you can do (for minimizing expected 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 $\mathrm{Var}(y) = \mathbb{E}y^2 - (\mathbb{E}y)^2$.]

Solution:

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

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 $\mathbb{E}\ell(a^*, y) = (\mathbb{E}y)^2 - 2(\mathbb{E}y)^2 + \mathbb{E}y^2 = \mathrm{Var}(y)$.

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 we 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. (Hint: There is really nothing to do here except write down the answer, based on the previous question. But make sure you understand what's happening...)

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. In math, we mean

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

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

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

22

for any f. Explain why this is true. (Hint: Law of iterated expectations.)

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

4.2 (OPTIONAL) Median Loss

1. (Optional) Show that for the absolute loss $\ell(\hat{y}, y) = |y - \hat{y}|$, $f^*(x)$ 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 $\mathbb{P}(y \geq m) \geq \frac{1}{2}$ and $\mathbb{P}(y \leq m) \geq \frac{1}{2}$.] Note: This loss function leads to "median regression". There are other loss functions that lead to "quantile regression" for any chosen quantile. (For partial credit, you may assume that the distribution of $y \mid x$ is discrete or continuous. For full credit, no assumptions about the distribution.)

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 $m > b$.

For $m < b$,

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

Taking expectations yields
$$\mathbb{E}\left(|Y - b| - |Y - m|\right) \geq \underbrace{(b - m)}_{>0}\underbrace{\left\{\mathbb{P}\left(Y \leq m\right) - \mathbb{P}\left(Y > m\right)\right\}}_{\geq 0 \text{ by definition of median}}$$
$$\geq 0$$

For $m > b$,

23

If $Y \geq m$, then

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

If $Y < m$, then

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

Putting this together, we get:

$$|Y - b| - |Y - m| \geq (m - b) \, 1(Y \geq m) - (m - b) \, 1(Y < m)$$
$$= (m - b) \, [1(Y \geq m) - 1(Y < m)]$$

Taking expectations yields

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

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