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

Nyutian Long

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

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 i^{th} element of μ and Σ_{ij} to denote the element at the i^{th} 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. $\|x\|_2$ represents the ℓ_2-norm of vector x.

   $\mathbb{E}\|x\|_2^2 = Var(\|x\|_2) + (\mathbb{E}\|x\|_2)^2 = \Sigma_{i=1}^n Var(x_i) + \mu^T\mu = tr(\Sigma) + \mu^T\mu$
   $\mathbb{E}\|x-y\|_2^2 = \mathbb{E}\|x\|_2^2 - 2\mathbb{E}\|x\|_2\|y\|_2 + \mathbb{E}\|y\|_2^2 = tr(\Sigma) + \mu^T\mu - 2\mu^T\mu + tr(\Sigma) + \mu^T\mu = 2tr(\Sigma)$

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.
   $X_i \sim \mathcal{N}(\mu_i, \sigma_{ii}), X_j \sim \mathcal{N}(\mu_j, \sigma_{jj})$ where σ_{ii} and σjj are the (i,i) and (j,j) entries in Σ.
   $Z = \alpha_i X_i + \alpha_j X_j \sim \mathcal{N}(\alpha_i\mu_i + \alpha_j\mu_j, \alpha_i^2\sigma_{ii} + 2\alpha_1\alpha_2\sigma_{X_iX_j} + \alpha_j^2\sigma_{jj})$.

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

2.2 Linear Algebra

(a) 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?
   By rank-nullity theorem, $rank(A) + null(A) = d \Rightarrow null(A) = dim(S_a) = d - k$.

(b) 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

$$x^* = \underset{x\in S_v}{\operatorname{argmin}} \|w - x\|_2,$$

   where $\|w - x\|_2$ is the Euclidean distance between w and x. Express x^* as a function of $v_1, v_2, \ldots, v_k$ and w.
   Since S_v is a k dimensional subspace in R^d, $v_1, v_2, \ldots, v_k, \ldots v_d$ would form an orthonormal basis of R^d. And w could be decomposed to its projection in $S_v, w|_{S_v}$ and $w - w|_{S_v}$. So $\underset{x\in S_v}{\operatorname{argmin}} \|w - x\|_2$ would be the projection of w onto S_v.

(c) (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.

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.

```
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
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
    """
    feature_max=np.amax(train, axis=0)
    feature_min=np.amin(train, axis=0)
    diff=feature_max-feature_min
    index=np.where(diff == 0)[0]
    train_new=np.delete(train, index, axis=1)
    test_new=np.delete(test, index, axis=1)
    feature_max_new=np.amax(train_new, axis=0)
    feature_min_new=np.amin(train_new, axis=0)
    train_normalized = (train_new-feature_min_new)/(feature_max_new-feature_min_new)
    test_normalized = (test_new-feature_min_new)/(feature_max_new-feature_min_new)
    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

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

[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.

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

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

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 $\theta, x \in \mathbf{R}^{d+1}$.

(a) 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.]

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

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

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

(c) 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.]

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

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

$$\theta' = \theta - \eta \nabla J(\theta; h) = \theta - \frac{2\eta}{m}(X\theta - y)^T x$$

(e) 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):
    """
    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 = X.shape[0]
    preds=np.dot(X,theta)
    loss = np.dot(preds-y,preds-y)/num_instances
    return loss
```

(f) 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):
    """
    Compute the gradient of the average square loss (as defined in compute_square_loss), at the point th

    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
    preds=np.dot(X,theta)
    grad=2.0/(X.shape[0])*np.dot(preds-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

[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.

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.

(a) 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.

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.

(a) Complete `batch_gradient_descent`.

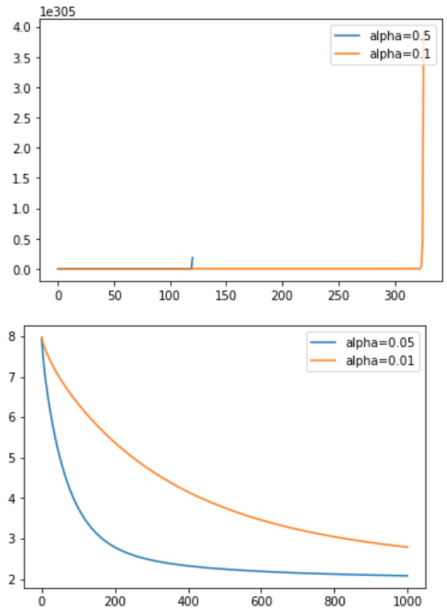
```
def batch_grad_descent(X, y, alpha, num_step, grad_check=False):
    """
    In this question you will implement batch gradient descent to
    minimize the average square loss objective.

    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_featur
                     for instance, theta in step 0 should be theta_hist[0], theta in step (num_step)
        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
    i=0
    while i < num_step+1:
        theta_hist[i]=theta
        loss_hist[i] = compute_square_loss(X, y, theta)
        theta = theta - alpha*compute_square_loss_gradient(X, y, theta)
        i=i+1
    return theta_hist, loss_hist
```

(b) 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

[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.

[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.

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.



Step sizes 0.5 and 0.1 are too large, making the average loss function diverge. With step sizes 0.05 and 0.01, the average loss function converges, and it converges faster at step size 0.05 because the gradient could descent with larger magnitude. A too large step size will make loss function diverge, and a too small step size will make loss function converge slowly.

(c) (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.)

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.

(a) 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.)

$$\nabla J(\theta) = \frac{2}{m}(\theta^T x - y)^T x + 2\lambda\theta$$

$$\theta' = \theta - \eta\nabla J(\theta) = \theta - \eta(\frac{2}{m}(\theta^T x - y)^T x + 2\lambda\theta)$$

(b) Implement `compute_regularized_square_loss_gradient`.

```
def compute_regularized_square_loss_gradient(X, y, theta, 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
    preds=np.dot(X,theta)
    grad=2.0/(X.shape[0])*np.dot(preds-y, X)+2*lambda_reg*theta
    return grad
```

(c) Implement `regularized_grad_descent`.

```
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_feat
                     for instance, theta in step 0 should be theta_hist[0], theta in step (num_ste
        loss hist - the history of average square loss function without the regularization term, 1
    """
    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
    i=0
    while i < num_step+1:
        theta_hist[i]=theta
        loss_hist[i] = compute_square_loss(X, y, theta)
        theta = theta - alpha*compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)
        i=i+1
    return theta_hist, loss_hist
```

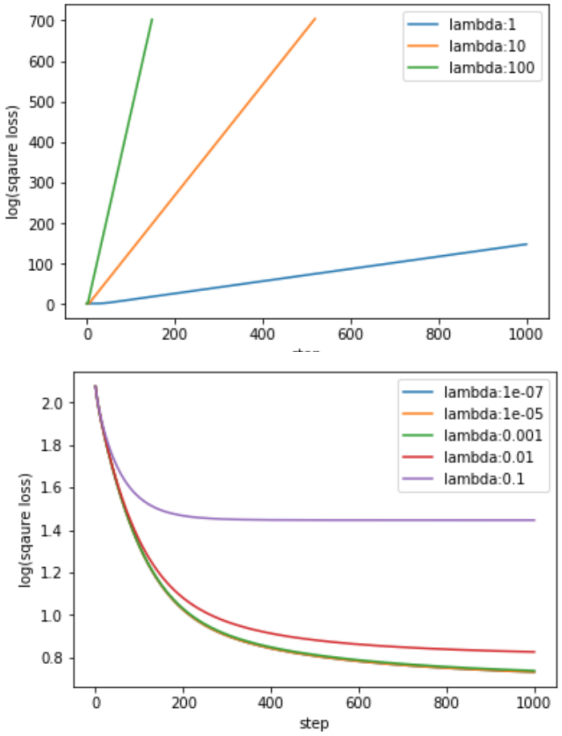
(d) For regression problems, we may prefer to leave the bias term unregularized. One approach is to change $J(\theta)$ so that the bias is separated out from the other parameters and left unregularized. Another approach that can achieve approximately the same thing is to use a very large number B, rather 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).

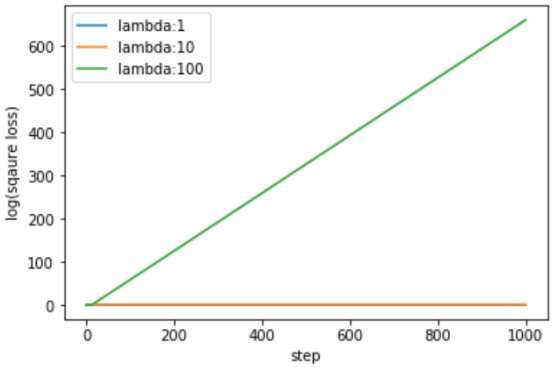
Because the bias term is the product of B and θ, which is adjusted by changes in θ and B. For large B, the magnitude of theta adjusting in regularization is small and the regularization terms of bias is small compared to the entire regularization.

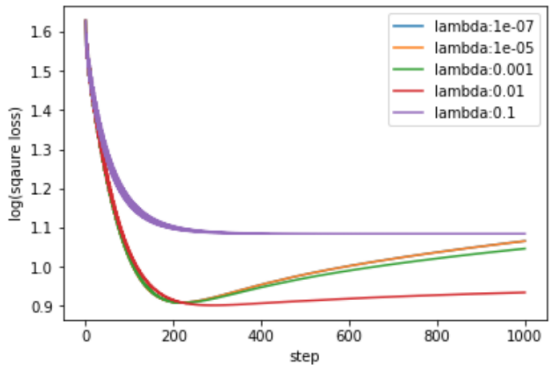
(e) (Optional) Develop a formal statement of the claim in the previous problem, and prove the statement.

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

(g) 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, $\lambda \in \{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\}$. 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. [If you like, you may use sklearn to help with the hyperparameter search.]







I chose step size = 0.05 here because it enabled fast convergence in the previous question.
The first two plots are loss for training set over $\lambda \in \left\{10^{-7}, 10^{-5}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\right\}$.
The third and fourth plots are for loss of test set over $\lambda \in \left\{10^{-7}, 10^{-5}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\right\}$.

(h) What θ would you select for deployment and why?
I would select $\theta_{\lambda=0.01}$ after the 400th step. Among $\lambda \in \left\{10^{-7}, 10^{-5}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\right\}$, $\lambda = 0.01$ gives the smallest loss on the test set. For θ, it converges well after the 400th iteration, so any θ after the 400th iteration will work well.

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.

(a) Show that the objective function

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

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.

$$\text{Let } f_i = (h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta, \text{ the two expressions are equal.}$$

(b) 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 $\mathbb{E}[\nabla f_i(\theta)] = \nabla J(\theta)$ for any 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 $\mathbb{E}[\nabla f_i(\theta)]$...)

$$\begin{aligned}
\mathbb{E}[\nabla f_i(\theta)] &= \mathbb{E}[2(h_\theta(x_i) - y_i)x_i + 2\lambda\theta] \\
&= \frac{2}{m}\Sigma_{i=1}^{m}\mathbb{E}((h_\theta(x_i) - y_i)x_i + \lambda\theta) \\
&= \mathbb{E}(\frac{2}{m}\Sigma_{i=1}^{m}(h_\theta(x_i) - y_i)x_i + 2\lambda\theta) \\
&= \nabla J(\theta)
\end{aligned}$$

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

$$\begin{aligned}
&\text{With ridge regression, } h_\theta(x) = \theta^T x \\
&\nabla f_i(\theta) = 2(\theta^T x_i - y_i)^T x_i + 2\lambda\theta \\
&\theta = \theta - 2\eta((\theta^T x_i - y_i)^T x_i + \lambda\theta)
\end{aligned}$$

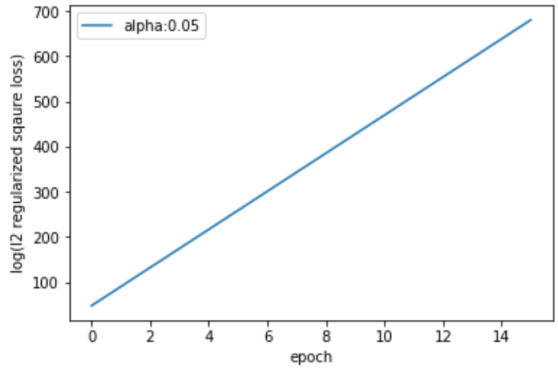
(d) Implement `stochastic_grad_descent`. (Note: You could potentially generalize the code you wrote for batch gradient to handle minibatches of any size, including 1, but this is not necessary.)

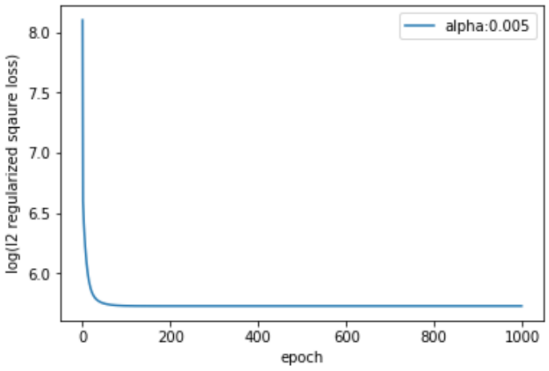
```python
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2, num_epoch=1000, c=0.1):
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) #Initialize theta
    theta_hist = np.zeros((num_epoch, num_instances, num_features)) #Initialize theta_hist
    loss_hist = np.zeros((num_epoch, num_instances)) #Initialize loss_hist
    s=np.arange(num_instances)
    np.random.shuffle(s)
    X = X[s]
    y = y[s]
    for i in range(num_epoch):
        for j in range(num_instances):
            if alpha=='c/sqrt(t)':
                step_size = c/np.sqrt(i*j+1.0)
            elif alpha=='c/t':
                step_size = c/(i*j+1.0)
            else:
                step_size = alpha
            theta_hist[i][j]= theta
            pred = np.dot(X[j], theta)
            f_j = np.dot(pred-y[j], pred-y[j])+np.dot(theta, theta)*lambda_reg
            loss_hist[i][j] = f_j+loss_hist[i][j-1]
            theta = theta - 2*step_size*(np.dot(pred-y[j], X[j])+lambda_reg*theta)
    return theta_hist, loss_hist
```

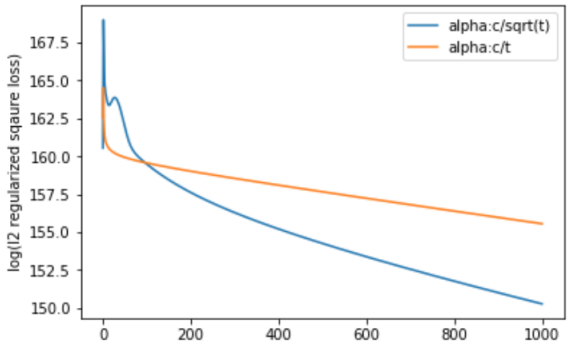
(e) Use SGD to find θ_λ^* that minimizes the ridge regression objective for the λ and B that you selected in the previous problem. (If you could not solve the previous problem, choose $\lambda = 10^{-2}$ and $B = 1$). Try a few fixed step sizes (at least try $\eta_t \in \{0.05, .005\}$. Note that SGD 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.







The first two plots are for fixed step size $\alpha = 0.05$ and $\alpha = 0.005$. When $\alpha = 0.05$, the loss function diverges. This is happening because the step is so large that we go over the minimum. The third are plots for $\alpha = \frac{c}{\sqrt{(t)}}$ and $\alpha = \frac{c}{t}$ when $c = 0.1$. They both

[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.

converges, and the loss function converges much faster when $\alpha = \frac{c}{\sqrt{(t)}}$. This is expected as $\frac{c}{\sqrt{t}}$ is bigger, making the loss function converge at a larger rate.

(f) (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?

4 Risk Minimization

4.1 Square Loss

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

$$\begin{aligned}
\mathbb{E}(a - y)^2 &= \mathbb{E}(a^2 - 2ay + y^2) = \mathbb{E}a^2 + \mathbb{E}y^2 - 2\mathbb{E}ay \\
&= Var(a) + (\mathbb{E}a)^2 + Var(y) + (\mathbb{E}y)^2 - 2a\mathbb{E}y \\
&= Var(a) + Var(y) + (a - \mathbb{E}y)^2 \\
&\text{(a is not a random variable, so } Var(a) = 0) \\
&= Var(y) + (a - \mathbb{E}y)^2 \\
&\text{Hence, } a^* = \mathbb{E}y, \text{ and the Bayes risk is } \mathrm{Var}(y).
\end{aligned}$$

(b) 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 $P_{\mathcal{X}\times\mathcal{Y}}$.

i. 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...)

Based on part a, $f^*(x) = \mathbb{E}\,(y|x)$

ii. 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

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

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

$$\begin{aligned} RHS =& \mathbb{E}[\mathbb{E}(f(x)-y)^2|x] = \mathbb{E}(Var(y|x)) + \mathbb{E}[(f(x)-\mathbb{E}(y|x))^2] \\ & \geq \mathbb{E}(Var(y|x)) \\ LHS =& \mathbb{E}(\mathbb{E}(y|x)-y)^2 = E(Var(y|x)) \\ & \text{Hence, } \mathbb{E}\left[(f^*(x)-y)^2\right] \leq \mathbb{E}\left[(f(x)-y)^2\right] \end{aligned}$$

4.2 (OPTIONAL) Median Loss

(a) 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.)