Homework_1

November 19, 2019

- Homepage of the course: MACHINE LEARNING
- Blog of My learning notes
- The solution is not necessarily correct.

1 Mathematical Fundamentals, Ridge Regression, Gradient Descent, and SGD

1.1 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`?

Comments

1.2 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.

1.2.1 Probability

Let $(X_1, X_2, \cdots, X_d)$ have a d-dimensional multivariate Gaussian distribution, with mean vector $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$, 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 $x, y \in \mathbb{R}^d$ 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.
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.
3. (Optional) Assume W and R are two Gaussian distributed random variables. Is $W + R$ still Gaussian? Justify your answer.

Solution

1. $$\mathbb{E}\|x\|_2^2 = \sum_{i=1}^{n} \mathbb{E}X_i^2 = \sum_{i=1}^{n} \left(Var(X_i) + (\mathbb{E}X_i)^2\right) = tr(\Sigma) + \|\mu\|_2^2$$

   Note that $\mathbb{E}[x-y] = 0, Var(x-y) = Var(x) + Var(y)$, then

   $$\mathbb{E}\|x-y\|_2^2 = 2 \cdot tr(\Sigma)$$

2. $\mathbb{E}[Z] = \alpha_i \mu_i + \alpha_j \mu_j$. However, μ_i, μ_j is not **iid**, which means we can't compute the variance by

   $$Var(Z) = \alpha_i^2 \Sigma_{ii} + \alpha_j^2 \Sigma_{jj} \tag{1}$$

   Actually, by definition, we can get $Var(Z) = \alpha_i^2 \Sigma_{ii} + \alpha_j^2 \Sigma_{jj} + 2\alpha_i \alpha_j \Sigma_{ij}$. But is it still normal? Sum of normally distributed random variables mentioned that the independence can be weakened to the assumption that X and Y are jointly, so

   $$Z \sim \mathcal{N}(\mathbb{E}Z, Var(Z))$$

   Actually, if X_i and X_j are independent, then $\Sigma_{ij} = cov(X_i, X_j) = 0$.
3. No! the joint distribution must be normal. For example, Let $W \sim \mathcal{N}(\mu, \sigma), R = mW$, where $m = 1$ with probability $1/2$, otherwise $m = -1$. we can prove that R is also normal, $W + R$ is not, however.

Simulation

```
In [43]: import numpy as np
         mean = np.array([1, 2, 3, 4, 5])
         cov = np.random.rand(5, 5)
         # Covariance matrix of the distribution.
         # It must be symmetric and positive-semidefinite for proper sampling.
         cov = np.dot(cov, cov.T)  # A*A^T must be positive-semidefinite
         X = np.random.multivariate_normal(mean, cov, size=40000)
```

Validation for $\mathbb{E}\|x\|_2^2 = tr(\Sigma) + \|\mu\|_2^2$

```
In [44]: print("bias 1 = ", np.sum(X**2,axis=1).mean() - (np.trace(cov) + np.sum(mean**2)))

bias 1 =  -0.0763274797266
```

Validation for $\mathbb{E}||x-y||_2^2 = 2 \cdot tr(\Sigma)$

```
In [46]: Y = np.random.multivariate_normal(mean, cov,size=40000)
         print("bias 2 = ", np.sum((X-Y)**2,axis=1).mean() - 2  * np.trace(cov))
```

```
bias 2 =  0.0811346722484
```

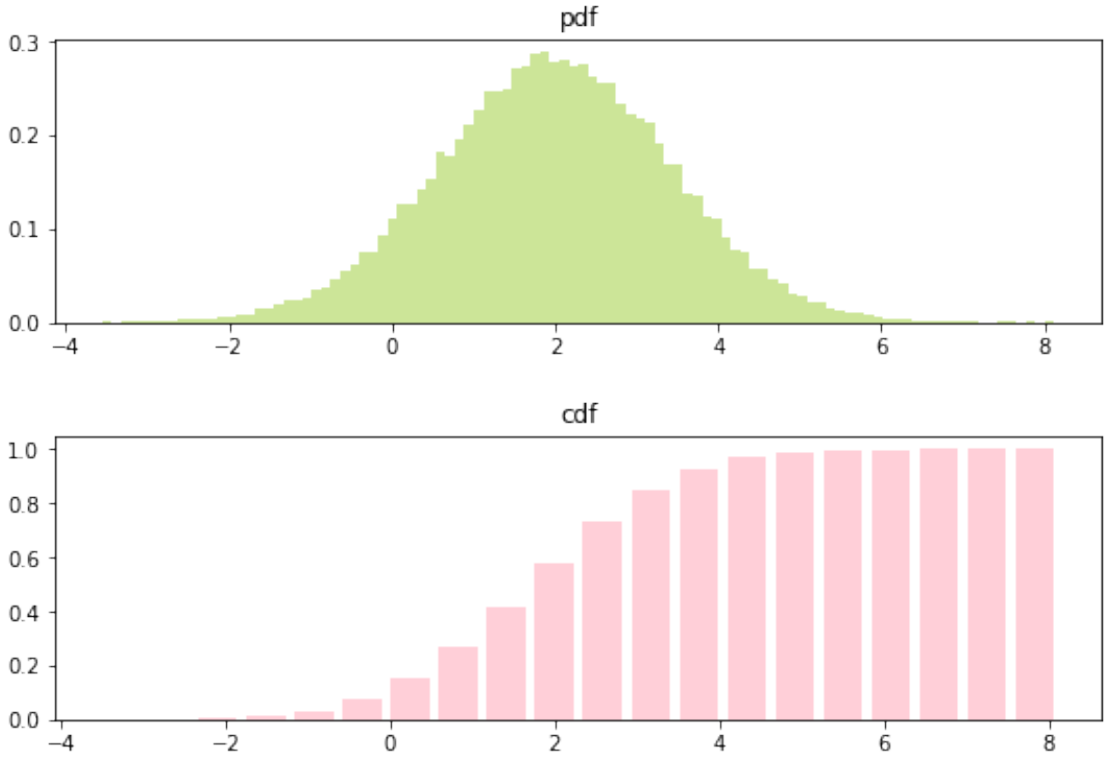
Validation for question 2. Let $i = 1, j = 3, \alpha_1 = \alpha_2 = 0.5$, note that $\mu = (1,2,3,4,5)$, we have $\mathbb{E}[Z] = 2$

```
In [5]: Var_Z = 0.25 * cov[0,0] + 0.25 * cov[2,2] + 0.5 * cov[0,2]
        print("True Variance: ", Var_Z)
        Z = 0.5 * X[:,0] + 0.5 * X[:,2]
        Var_Z_sample = (Z**2).mean() - Z.mean() ** 2
        print("Sample Variance: ", Var_Z_sample)
```

```
True Variance:  2.02996535883
Sample Variance:  2.02323252829
```

Visualization of normal distribution of Z

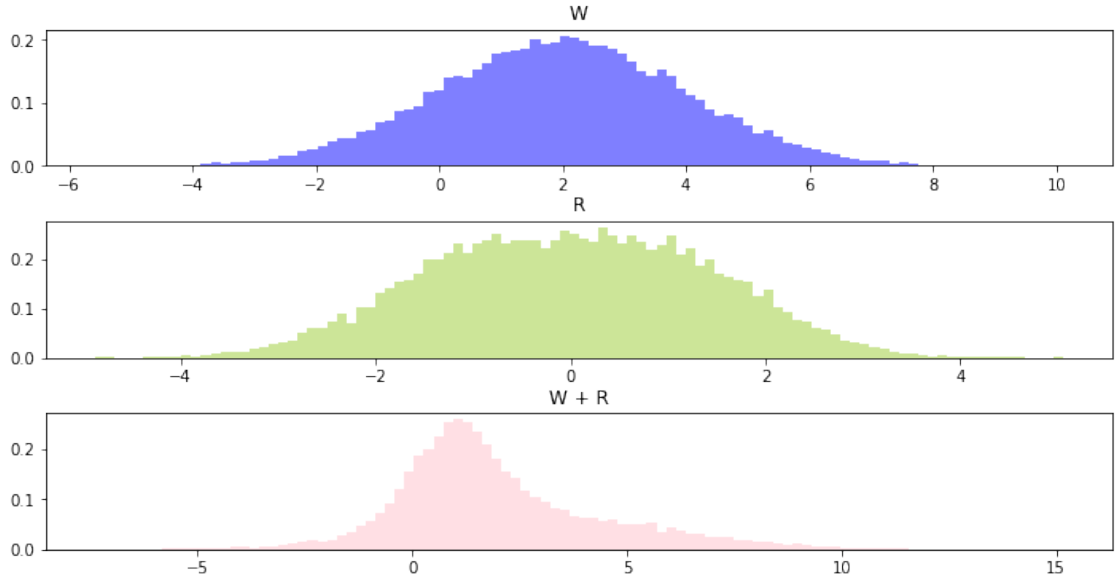
```
In [6]: import matplotlib.pyplot as plt
        %matplotlib inline
        fig, (ax0, ax1) = plt.subplots(nrows=2, figsize=(9, 6))
        ax0.hist(Z, 100, density=1, histtype='bar', facecolor='yellowgreen', alpha=0.5)
        ax0.set_title('pdf')
        ax1.hist(Z, 20, density=1, histtype='bar', facecolor='pink',
                 alpha=0.75, cumulative=True, rwidth=0.8)
        ax1.set_title("cdf")
        fig.subplots_adjust(hspace=0.4)
        plt.show()
```



Counterexample for question 3

```
In [7]: W = np.random.normal(2,2,size=20000)
        m = np.random.binomial(1,0.5,20000) - 0.5
        R = m * W
```

```
In [8]: fig,(ax0,ax1,ax2) = plt.subplots(nrows=3,figsize=(12,6))
        ax0.hist(W,100,density=1,histtype='bar',facecolor='blue',alpha=0.5)
        ax0.set_title('W')
        ax1.hist(R,100,density=1,histtype='bar',facecolor='yellowgreen',alpha=0.5)
        ax1.set_title('R')
        ax2.hist(W + R,100,density=1,histtype='bar',facecolor='pink',alpha=0.5)
        ax2.set_title("W + R")
        fig.subplots_adjust(hspace=0.4)
        plt.show()
```



1.2.2 Linear Algebra

1. Let A be a $d \times d$ matrix with rank k. Consider the set $S_A := \{x \in \mathbb{R}^d | Ax = 0\}$. What is the dimension of S_A?
2. Assume S_v is a k dimensional subspace in $\mathbb{R}^d$ and $v_1, v_2, \cdots, v_k$ form an orthonormal basis of S_v. Let w be an arbitrary vector in $\mathbb{R}^d$. Find

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

   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.
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

1. $\dim(S_A) = Ker(A) = d - rank(A) = d - k$
2. Actually, x^* is the orthogonal projection of w. we have

$$< w - x^*, v_i > = 0, i = 1, 2, \cdots, k$$

   At the same time, $x^* = \sum_{i=1}^{k} \alpha_i v_i$, then we can get

$$x^* = \sum_{i=1}^{k} < w, v_i > \cdot v_i$$

3. $x^* = V\alpha$, where $V = [v_1, v_2, \cdots, v_k]_{d\times k}, \alpha_i =< w, v_i >$, so

$$x^* = VV^T w$$

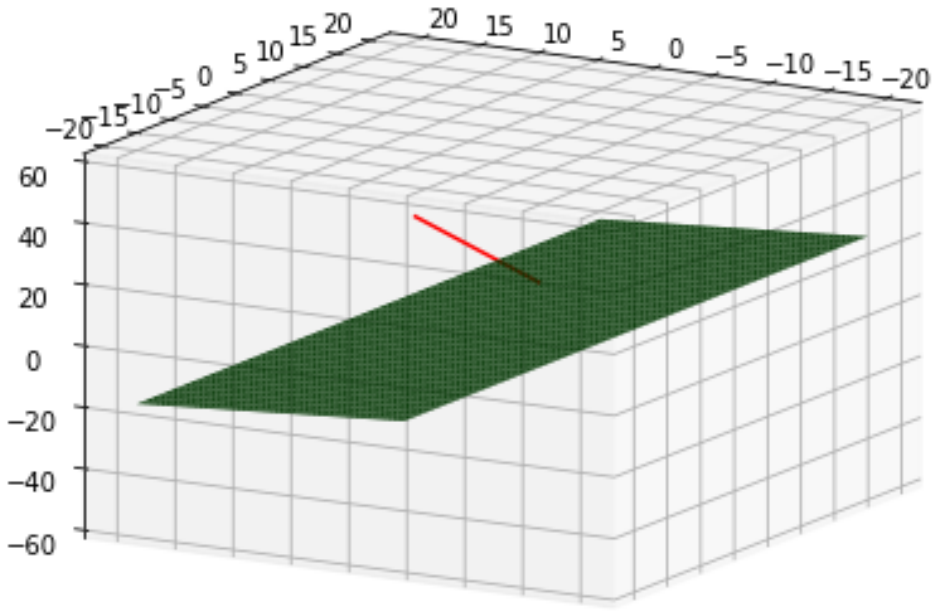
which means $M = VV^T$. More over, $w = (VV^T)^{-1}V\alpha$. We can easily prove that $M^2 = M$. Let λ be the eigenvalue then $Mx = \lambda x$, multiply M to both left and right we get

$$Mx = M^2 x = \lambda Mx$$

- If $Mx = 0$, then $\lambda = 0$, and $\dim\{x \mid Mx = 0\} = d - rank(M) = d - rank(V) = d - k$.
- If $Mx \neq 0$, then $\lambda = 1$. It's obvious that $Mv_i = v_i, i = 1, 2, \cdots, k$.

Simulation Considering the plane $2x + y + z = 0$ in the euclidean space $\mathbb{R}^3$.

```
In [9]: from scipy.linalg import *
        a = np.array([[0,-1],[1,0],[-1,2]])
        V = orth(a)
        M = np.dot(V,V.T)
        eigen_value, vectors = eig(M)

In [10]: eigen_value.real

Out[10]: array([ -1.11022302e-16,   1.00000000e+00,   1.00000000e+00])

In [11]: w = np.array([7.5,0, 30])
         projection = np.dot(M,w)

In [12]: from mpl_toolkits.mplot3d.axes3d import Axes3D
         fig = plt.figure()
         axes3d = Axes3D(fig,proj_type="ortho", azim=60,elev=-20)
         x = np.linspace(-20,20,100)
         y = np.linspace(-20,20,100)
         X,Y = np.meshgrid(x,y)
         Z = -2*X - Y
         x, y, z = [w[0], projection[0]], [w[1], projection[1]], [w[2], projection[2]]
         axes3d.plot_surface(X,Y,Z, color="green")
         axes3d.plot(x, y, z, c='r')

Out[12]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f77d6d134a8>]
```



1.3 Linear Regression

1.3.1 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 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

```
In [13]: 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 num5py array of size (num_instances, num_features)

        Returns:
            train_normalized - training set after normalization
            test_normalized - test set after normalization
        """
        num_instances, num_features = train.shape
        # discard features that are constant
        index = np.nonzero(np.all(X_train == X_train[0, :], axis=0))
        train = np.delete(train, index, axis=1)
        test = np.delete(test, index, axis=1)
        # Min-Max normalization
        div = np.max(X_train, axis=0, keepdims=True) - \
            np.min(X_train, axis=0, keepdims=True)
        minus = np.min(X_train, axis=0, keepdims=True)
        train = (train - minus) / div
        test = (test - minus) / div
        return train, test
```

In [14]:
```
from sklearn.model_selection import train_test_split
import pandas as pd

# Loading the dataset
print('loading the dataset')

df = pd.read_csv("data.csv", delimiter=',')
X = df.values[:, :-1]
y = df.values[:, -1]

print('Split into Train and Test')
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=100, random_state=10)
print("Scaling all to [0, 1]")
X_train, X_test = feature_normalization(X_train, X_test)

# Add bias term
X_train = np.hstack((X_train, np.ones((X_train.shape[0], 1))))
X_test = np.hstack((X_test, np.ones((X_test.shape[0], 1))))
```

```
loading the dataset
Split into Train and Test
Scaling all to [0, 1]
```

1.3.2 Gradient Descent Setup

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

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

for $\theta, x \in \mathbb{R}^d$, 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 $(x_1, y_1), \ldots, (x_m, y_m) \in \mathbb{R}^d \times \mathbb{R}$ 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** $\theta, x \in \mathbb{R}^{d+1}$.

1. Let $X \in \mathbb{R}^{m \times (d+1)}$ be the **design matrix**, where the i'th row of X is x_i. Let $y = (y_1, \ldots, y_m)^T \in \mathbb{R}^{m \times 1}$ be the `response`. Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign.
2. Write down an expression for the gradient of J (again, as a matrix/vector expression, without using an explicit summation sign).
3. In our search for a θ that minimizes J, suppose we take a step from θ to $\theta + \eta h$, where $h \in \mathbb{R}^{d+1}$ 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.
4. Write down the expression for updating θ in the gradient descent algorithm. Let η be the step size.
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.
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

1. $$J(\theta) = \frac{1}{m} \|X\theta - y\|_2^2$$
2. $$\frac{\mathrm{d}J(\theta)}{\mathrm{d}\theta} = \frac{1}{m} \frac{\mathrm{d}(X\theta - y)^T}{\mathrm{d}\theta} \cdot 2(X\theta - y) = \frac{2}{m} X^T (X\theta - y)$$

3. We always set $h = -\nabla_\theta J(\theta)$, then we get

$$J(\theta + \eta h) - J(\theta) \approx -\eta(\nabla_\theta J(\theta))^T \theta \approx -\frac{2}{m}\eta(\theta^T X^T - y^T)X\theta$$

4.

$$\theta_{i+1} = \theta_i - \eta \nabla_\theta J(\theta_i) = \theta_i - \frac{2}{m}\eta X^T(X\theta_i - y)$$

In [15]:
```
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
    """
    num_instances = X.shape[0]
    bias = np.dot(X, theta) - y.reshape(num_instances, 1)
    loss = np.sum(bias ** 2) / num_instances
    return loss
```

In [16]:
```
theta_init = np.random.rand(X_train.shape[1], 1)
compute_square_loss(X_train, y_train, theta_init)
```

Out[16]: 222.88995536215657

In [17]:
```
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)
    """
    grad = 2 * np.dot(X.T, np.dot(X, theta) - y.reshape(X.shape[0], 1)) / X.shape[0]
    return grad
```

In [18]:
```
compute_square_loss_gradient(X_train, y_train, theta_init).shape
```

Out[18]: (49, 1)

1.3.3 Gradient Checker

1.3.4 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`. 2. Now let's experiment with the step size. Note that **if the step size is too large, gradient descent may not converge**. * 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

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. 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

```
In [49]: 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, 1))  # Initialize theta

             for step in range(num_step):
```

```
                loss = compute_square_loss(X, y, theta)
                theta_hist[step] = theta.T
                loss_hist[step] = loss
                theta = theta - alpha * compute_square_loss_gradient(X, y, theta)

            theta_hist[num_step] = theta.T
            loss_hist[num_step] = compute_square_loss(X, y, theta)
            return theta_hist, loss_hist
```

```
In [50]: theta_hist, loss_hist = batch_grad_descent(
             X_train, y_train, alpha=0.01, num_step=1000)
```

```
In [51]: def R_square(y, pred):
             y = y.reshape([len(y), 1])
             pred = pred.reshape([len(pred), 1])
             SS_tol = (y ** 2).mean() - (y.mean()) ** 2
             SS_res = sum((y - pred) ** 2) / y.shape[0]
             return 1 - SS_res / SS_tol
```

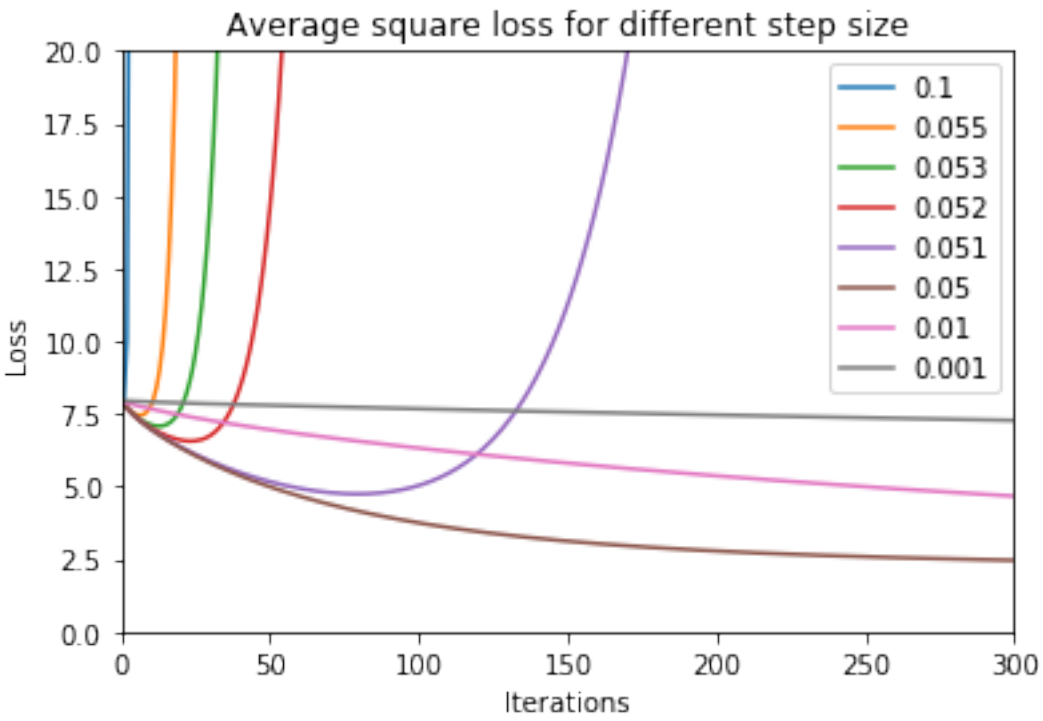
```
In [52]: pred = np.dot(X_train,theta_hist[-1])
         print("R^2:" , R_square(y_train, pred))
```

```
R^2: [ 0.63683116]
```

```
In [23]: alpha_list = [0.1, 0.055, 0.053, 0.052, 0.051, 0.05, 0.01, 0.001]
         loss_hist_alpha = [batch_grad_descent(
             X_train, y_train, alpha=alpha, num_step=1000)[1] for alpha in alpha_list]
```

```
/home/equation/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:16: RuntimeWarning:
  app.launch_new_instance()
/home/equation/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:14: RuntimeWarning:
```

```
In [24]: x = np.arange(0,1001)
         for i in range(len(alpha_list)):
             plt.plot(x,loss_hist_alpha[i],label=str(alpha_list[i]))

         plt.title('Average square loss for different step size')
         plt.xlabel('Iterations')
         plt.ylabel('Loss')
         plt.ylim(0, 20)
         plt.xlim(0, 300)
         plt.legend()
         plt.show()
```



In [25]:
```
import time
```

```
def batch_grad_descent_backtracking(X, y, num_step=1000, t=0.4, beta=0.5,
                                    grad_check=False):
    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
    alpha_hist = np.zeros(num_step+1)
    theta = np.zeros((num_features, 1))  # Initialize theta
    com_grad_time = 0
    search_time = 0

    for step in range(num_step):
        loss = compute_square_loss(X, y, theta)
        theta_hist[step] = theta.T
        loss_hist[step] = loss
        alpha = 1.0
        start = time.clock()
        grad = compute_square_loss_gradient(X, y, theta)
        end = time.clock()
        com_grad_time += (end - start)
        start = time.clock()
        while compute_square_loss(X, y, theta - alpha * grad) > \
```

```
                compute_square_loss(X, y, theta) + t * alpha * np.dot(grad.T, -grad):
                    alpha = alpha * beta

                end = time.clock()
                search_time += (end - start)

                theta = theta - alpha * grad
                alpha_hist[step] = alpha

            theta_hist[num_step] = theta.T
            alpha_hist[num_step] = alpha
            loss_hist[num_step] = compute_square_loss(X, y, theta)

            print("Compute the gradient running time: " + str(com_grad_time))
            print("backtracking line search running time: " + str(search_time))
            return theta_hist, loss_hist, alpha_hist
```

```
In [26]: theta_hist, loss_hist, alpha_hist = batch_grad_descent_backtracking(
             X_train, y_train, num_step=1000)
```

```
Compute the gradient running time: 0.018154000000066617
backtracking line search running time: 0.19025799999996806
```

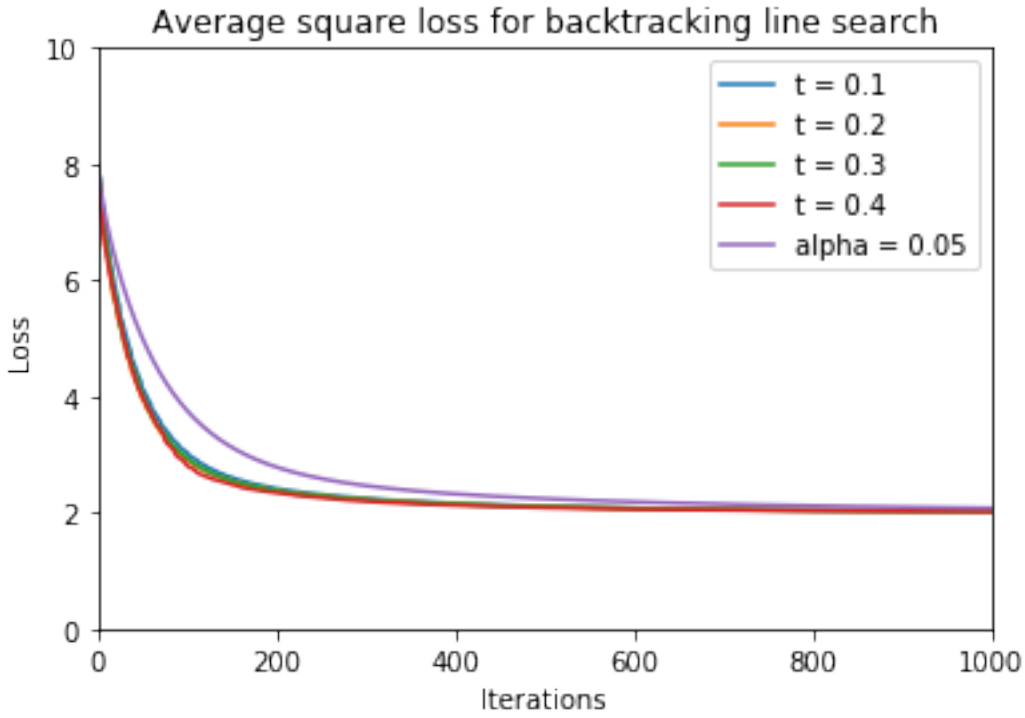
```
In [27]: _, loss_fixed_alpha = batch_grad_descent(
             X_train, y_train, alpha=0.05, num_step=1000)
```

```
In [28]: t_list = [0.1, 0.2, 0.3, 0.4]
         loss_backtracking = [batch_grad_descent_backtracking(
             X_train, y_train, t=t, num_step=1000)[1] for t in t_list]
```

```
Compute the gradient running time: 0.023590000000044853
backtracking line search running time: 0.25139299999993625
Compute the gradient running time: 0.022660000000058744
backtracking line search running time: 0.22858500000004867
Compute the gradient running time: 0.01978599999998565
backtracking line search running time: 0.2026590000000965
Compute the gradient running time: 0.022147999999983625
backtracking line search running time: 0.23027200000003845
```

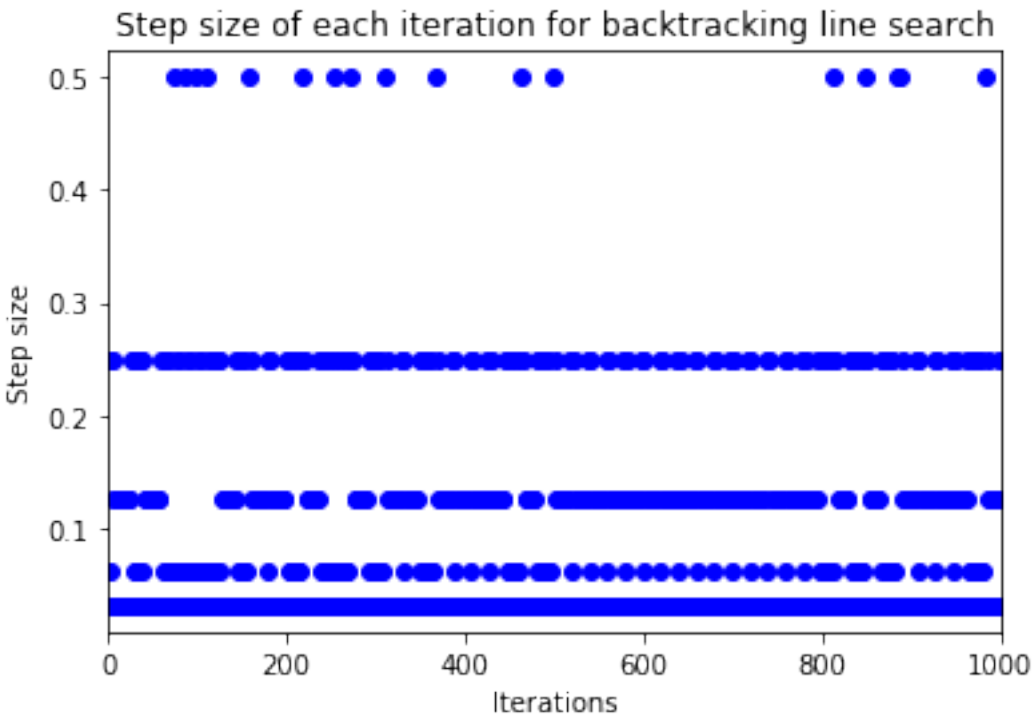
```
In [29]: x = np.arange(0,1001)
         for i in range(len(t_list)):
             plt.plot(x,loss_backtracking[i],label= "t = " + str(t_list[i]))

         plt.plot(x,loss_fixed_alpha,label= "alpha = 0.05" )
         plt.title('Average square loss for backtracking line search')
         plt.xlabel('Iterations')
         plt.ylabel('Loss')
```

```
plt.ylim(0, 10)
plt.xlim(0, 1000)
plt.legend()
plt.show()
```



In [30]:
```
alpha_hist = batch_grad_descent_backtracking(
    X_train, y_train, t=0.4, num_step=1000)[2]
x = np.arange(0, 1001)

plt.plot(x, alpha_hist, 'bo')
plt.title('Step size of each iteration for backtracking line search')
plt.xlabel('Iterations')
plt.ylabel('Step size')
plt.xlim(0, 1000)
plt.show()
```

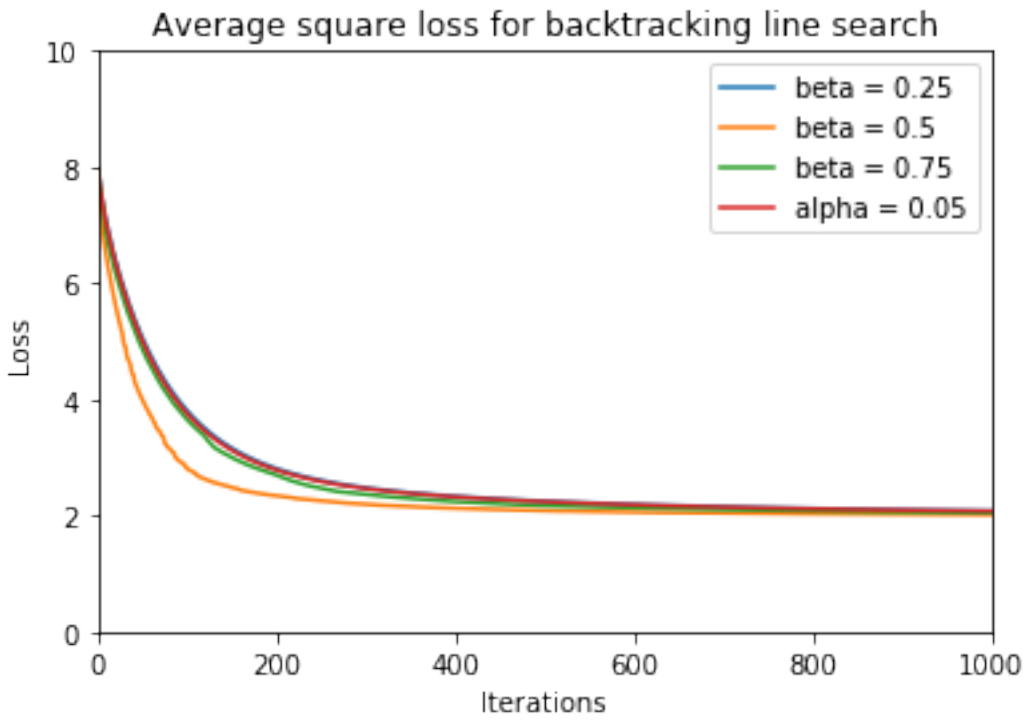
```
Compute the gradient running time: 0.01831799999999717
backtracking line search running time: 0.1899860000000011
```




```
In [31]: beta_list = [0.25, 0.5, 0.75]
         loss_backtracking = [batch_grad_descent_backtracking(
             X_train, y_train, t=0.4, beta=beta, num_step=1000)[1] for beta in beta_list]
```

```
Compute the gradient running time: 0.02178700000001399
backtracking line search running time: 0.1527189999999905
Compute the gradient running time: 0.021406999999960874
backtracking line search running time: 0.22226199999993312
Compute the gradient running time: 0.021609999999903096
backtracking line search running time: 0.4933860000000294
```

```
In [32]: x = np.arange(0,1001)
         for i in range(len(beta_list)):
             plt.plot(x,loss_backtracking[i],label= "beta = " + str(beta_list[i]))

         plt.plot(x,loss_fixed_alpha,label= "alpha = 0.05" )
         plt.title('Average square loss for backtracking line search')
         plt.xlabel('Iterations')
         plt.ylabel('Loss')
         plt.ylim(0, 10)
         plt.xlim(0, 1000)
         plt.legend()
         plt.show()
```



1.4 Ridge Regression

Linear Regression with ℓ_2 regularization or weight decay.

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)
2. Implement `compute_regularized_square_loss_gradient`
3. Implement `regularized_grad_descent`
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).

5. (Optional) Develop a formal statement of the claim in the previous problem, and prove the statement.
6. (Optional) Try various values of B to see what performs best in test.
7. Now fix $B = 1$. Choosing a reasonable step-size (or using backtracking line search), find the θ_λ^* that minimizes $J(\theta)$ over a range of λ. You should plot the training 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.

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

1.4.1 Solution

1.

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

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

Equation 2 is called weight-decay in deep learning, where

$$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i} - \eta\lambda w_i$$

corresponding to the optimization problem

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

```
In [33]: 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)
             """
```

```
        grad = 2 * np.dot(X.T, np.dot(X, theta) -
                          y.reshape(X.shape[0], 1)) / X.shape[0] + 2 * lambda_reg * theta
        return grad
```

In [34]:
```
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_ste
                     for instance, theta in step 0 should be theta_hist[0], theta in
        loss hist - the history of average square loss function without the regulariz
    """
    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, 1))  # Initialize theta

    for step in range(num_step):
        loss = compute_square_loss(X, y, theta)
        theta_hist[step] = theta.T
        loss_hist[step] = loss
        theta = theta - alpha * \
            compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)

    theta_hist[num_step] = theta.T
    loss_hist[num_step] = compute_square_loss(X, y, theta)
    return theta_hist, loss_hist
```

In [35]:
```
theta_hist, loss_fixed_alpha = regularized_grad_descent(
    X_train, y_train, alpha=0.05, num_step=1000)
```

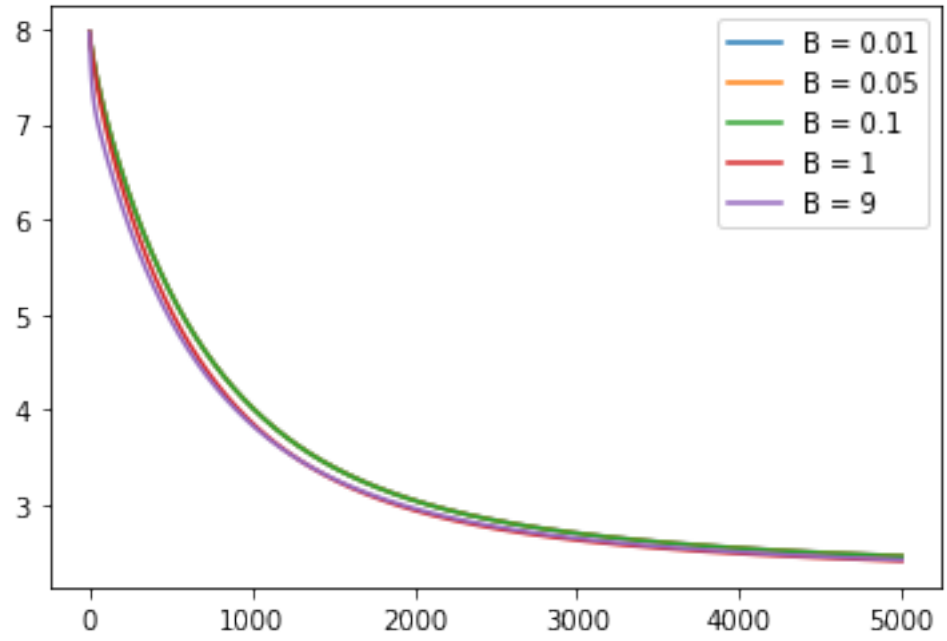
Try various values of B to see what performs best in test.

In [36]:
```
B_list = [0.01, 0.05, 0.1, 1, 9]
num_step = 5000
x = np.arange(0, num_step + 1)
for B in B_list:
    X_train[:, -1] = B * np.ones([X_train.shape[0], 1])[:, 0]
    X_test[:, -1] = B * np.ones([X_train.shape[0], 1])[:, 0]
    theta_hist, loss_fixed_alpha = regularized_grad_descent(
        X_train, y_train, alpha=0.005, lambda_reg=1e-2, num_step=num_step)
    test_err = compute_square_loss(
        X_test, y_test, theta_hist[-1].reshape([X_train.shape[1], 1]))
```

```
            print("Test loss of B = " + str(B) + " is " + str(test_err))
            plt.plot(x, loss_fixed_alpha, label="B = " + str(B))

        plt.legend()
```

```
Test loss of B = 0.01 is 2.45697156245
Test loss of B = 0.05 is 2.4566659192
Test loss of B = 0.1 is 2.45579424233
Test loss of B = 1 is 2.4923145059
Test loss of B = 9 is 2.51673821964
```

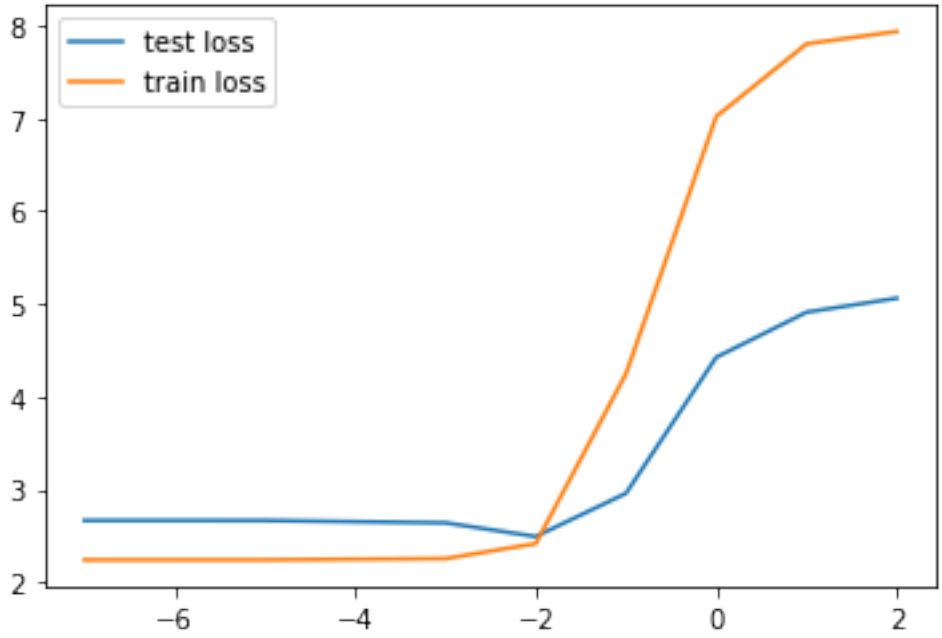
Out[36]: <matplotlib.legend.Legend at 0x7f77b9a81748>



In [37]:
```
# num_step = 1000
X_train[:, -1] = np.ones([1, X_train.shape[0]])
X_test[:, -1] = np.ones([1, X_train.shape[0]])
lambda_list = [-7, -6, -5, -3, -2, -1, 0, 1, 2]
train_loss = []
test_loss = []
for lamda in lambda_list:
    theta_hist, loss_fixed_alpha = regularized_grad_descent(
        X_train, y_train, alpha=0.005, lambda_reg=10 ** lamda, num_step=num_step)
    test_loss.append(compute_square_loss(
        X_test, y_test, theta_hist[-1].reshape([X_train.shape[1], 1])))
    train_loss.append(compute_square_loss(X_train, y_train,
                                          theta_hist[-1].reshape([X_train.shape[1], 1
```

```
In [38]: plt.plot(lambda_list,test_loss,label= "test loss")
         plt.plot(lambda_list,train_loss,label= "train loss")
         plt.legend()
```

```
Out[38]: <matplotlib.legend.Legend at 0x7f77b96099e8>
```



1.5 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 i th 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

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

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. 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 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 $\mathbb{E}\left[\nabla f_i(\theta)\right]$...) 3. Write down the update rule for θ in SGD for the ridge regression objective function.

1.5.1 Solution

1. Let
$$f_i(\theta) = \left(h_\theta\left(x_i\right) - y_i\right)^2 + m\lambda\theta^T\theta$$
which would conclude the proof.
2. Differentiate both sides of equation 25 and use the linearity of differentiation
$$\nabla J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(\theta)$$
Since the examples are chosen uniformly at random,
$$\mathbb{E}\left[\nabla f_i(\theta)\right] = \sum_{j=1}^{m} P(i=j) \times \nabla f_i(\theta) = \sum_{j=1}^{m} \frac{1}{m} \times \nabla f_i(\theta)$$
which would conclude the proof.
3. $\nabla f_i(\theta) = 2x_i(x_i^T\theta - y_i) + 2m\lambda\theta$, then
$$\theta_{i+1} = \theta_i - \eta\nabla f_i(\theta) = \theta_i - 2\eta x_i(x_i^T\theta_i - y_i) - 2\eta m\lambda\theta_i$$

```
In [39]: import math


         def GradientAtDatapoint(theta, datapoint, lable, lambda_reg, num_instances):
             return 2 * (np.dot(datapoint.T, theta) - lable) * datapoint + 2 * num_instances *


         def get_batchs(datasets_order, num_instances, batch_size):
             num_batches = num_instances // batch_size
             for i in range(num_batches):
                 yield datasets_order[i * batch_size: (i+1) * batch_size]


         # Stochastic gradient descent


         def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2, num_epoch=1000, batch
             """
```

```
    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
                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
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.zeros([num_features, 1])  # Initialize theta

    num_batches = num_instances // batch_size

    # Initialize theta_hist
    theta_hist = np.zeros((num_epoch, num_batches + 1, num_features))
    loss_hist = np.zeros((num_epoch, num_batches + 1))  # Initialize loss_hist
    # TODO
    iteration = 1
    for epoch in range(num_epoch):

        # generate initial subscript
        datasets_order = list(range(0, num_instances))
        np.random.shuffle(datasets_order)  # shuffle datasets
        batchs = get_batchs(datasets_order, num_instances, batch_size)
        for i, batch_order in enumerate(batchs):
            loss = compute_square_loss(X, y, theta)
            loss_hist[epoch, i] = loss
            theta_hist[epoch, i] = theta.T

            grad_matrix = np.array([GradientAtDatapoint(theta, X[index].reshape(
                [num_features, 1]), y[index], lambda_reg, num_instances) for index in
            grad = grad_matrix.reshape([batch_size, num_features]).mean(
                axis=0).reshape([num_features, 1])

            if sqrt_mode:
                alpha = C / math.sqrt(iteration)
            else:
                alpha = C / iteration

            theta = theta - alpha * grad
            iteration += 1
```

```
            loss = compute_square_loss(X, y, theta)
            loss_hist[epoch, num_batches] = loss
            theta_hist[epoch, num_batches] = theta.T
            return theta_hist, loss_hist
```

```
In [53]: X_train[:, -1] = 0.01 * np.ones([1, X_train.shape[0]])
         theta_hist, loss_hist = stochastic_grad_descent(
             X_train, y_train, num_epoch=1000, C=0.1, batch_size=1, lambda_reg=1e-4, sqrt_mode
```

1.6 Risk Minimization

1.6.1 Square Loss

1. Let y be a random variable with a known distribution, and consider the square loss function $\ell(a, y) = (a - y)^2$. We want to find the action a^* that has minimal risk. That is, we want to find
$$a^* = \arg\min_a \mathbb{E}\,(a - y)^2$$
where the expectation is with respect to y. Show that $a^* = \mathbb{E}y$, and the Bayes risk (i.e. the risk of a^*) is $Var(y)$. 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 $Var(y) = \mathbb{E}y^2 - (\mathbb{E}y)^2$.

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:
$$R\left(f^*\right) = \inf_f R(f)$$
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 $f^*(x) = \mathbb{E}[y|x]$, where the expectation is over y. As before, we assume we know the data-generating distribution $P_{\mathcal{X}\times\mathcal{Y}}$.

   - 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
$$f^*(x) = \arg\min_a \mathbb{E}\left[(a - y)^2|x\right]$$
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...)

- 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.)

Solution

1. **Proof:** Recall that $Var(y) = \mathbb{E}y^2 - (\mathbb{E}y)^2$, we have

$$\mathbb{E}\,(a - y)^2 = Var(a - y) + [\mathbb{E}(a - y)]^2 = Var(y) + (a - \mathbb{E}y)^2 \tag{3}$$

which means when $a^* = \mathbb{E}y$, the bayes risk of a^* has the minimum value $Var(y)$.

1.6.2 Median Loss

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.)