DS-GA 3001.007
Homework 2: Gradient Descent and Stochastic Gradient Descent

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 implement linear 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 starter code, which you can download from JupyterHub. 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" `https://leon.bottou.org/papers/bottou-tricks-2012`)
- 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?

2 Linear Regression

2.1 Feature Normalization

When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization (introduced in a later problem), features with larger values 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.

2.2 Gradient Descent Setup

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

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

for $\theta, x \in \mathbf{R}^d$, and we choose θ that minimizes the following "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 $\theta, x \in \mathbf{R}^{d+1}$.

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:
   $$J(\theta) = \frac{1}{m} \|X\theta - y\|^2$$

2. Write down an expression for the gradient of J (again, as a matrix/vector expression, without using an explicit summation sign).
   Solution:
   $$\nabla J(\theta) = \frac{2}{m} X^T (X\theta - y)$$

[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.

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:
$$\begin{aligned} J(\theta + \eta h) - J(\theta) &= J(\theta) + \eta \nabla J(\theta)^T h + O(\eta^2 \|h\|^2) - J(\theta) \\ &\approx \eta \nabla J(\theta)^T h \end{aligned}$$

4. Write down the expression for updating θ in the gradient descent algorithm. Let η be the step size.
Solution:
$$\theta = \theta - \eta \nabla J(\theta)$$

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.

2.3 Batch Gradient Descent[2]

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. 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.
Answer: The gradient descent diverges when $\alpha = 0{:}5$ or $\alpha = 0{:}1$. When converging, the one with a larger step size converges faster.

3. Implement backtracking line search. 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.)
Answer: Backtrack line search converges in less steps. It yields a slightly lower, though not significantly, average square loss after 1000 steps. It takes longer time than the fixed step size algorithm.

[2]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.

2.4 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

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

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:

$$f_i(\theta) = \left(h_\theta(x_i) - y_i\right)^2$$

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 linear regression. You can start by writing down an expression for $\mathbb{E}\left[\nabla f_i(\theta)\right]$...)
Solution:

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

Hence it is unbiased.

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

$$\begin{aligned}\theta^{'} &= \theta - \eta \nabla f_i(\theta) \\ &= \theta - \eta \nabla \left(\theta^T x_i - y_i\right)^2 \\ &= \theta - 2\eta \left(\theta^T x_i - y_i\right) x_i\end{aligned}$$

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.)

5. Use SGD to find θ that minimizes the linear regression objective. 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 4, 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.
   - 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[3] and see how it compares.

[3] 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.

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.

hw2

October 19, 2019

[ ]:
```
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
    """
    # TODO
    # Get the stats
    train_max = train.max(axis = 0)
    train_min = train.min(axis = 0)

    # Delete the features that have constant value
    equal_indicator = (train_max != train_min)

    train = train[:,equal_indicator]
    test = test[:,equal_indicator]
    train_max = train_max[equal_indicator]
    train_min = train_min[equal_indicator]

    # Normalize (uses array broadcasting http://wiki.scipy.org/
 ↪EricsBroadcastingDoc)
    train_normalized = (train - train_min) / (train_max - train_min)
    test_normalized = (test - train_min) / (train_max - train_min)

    return train_normalized, test_normalized
```

[ ]:
```
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
```

[ ]:
```
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
```

[ ]:
```
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.
    Let d be the number of features. Here we numerically estimate the
    gradient by approximating the directional derivative in each of
    the d coordinate directions:
    (e_1 = (1,0,0,...,0), e_2 = (0,1,0,...,0), ..., e_d = (0,...,0,1))
    The approximation for the directional derivative of J at the point
    theta in the direction e_i is given by:
    ( J(theta + epsilon * e_i) - J(theta - epsilon * e_i) ) / (2*epsilon).
    We then look at the Euclidean distance between the gradient
    computed using this approximation and the gradient computed by
    compute_square_loss_gradient(X, y, theta).  If the Euclidean
    distance exceeds tolerance, we say the gradient is incorrect.
    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)

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)
```

```
[ ]: 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
             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
```

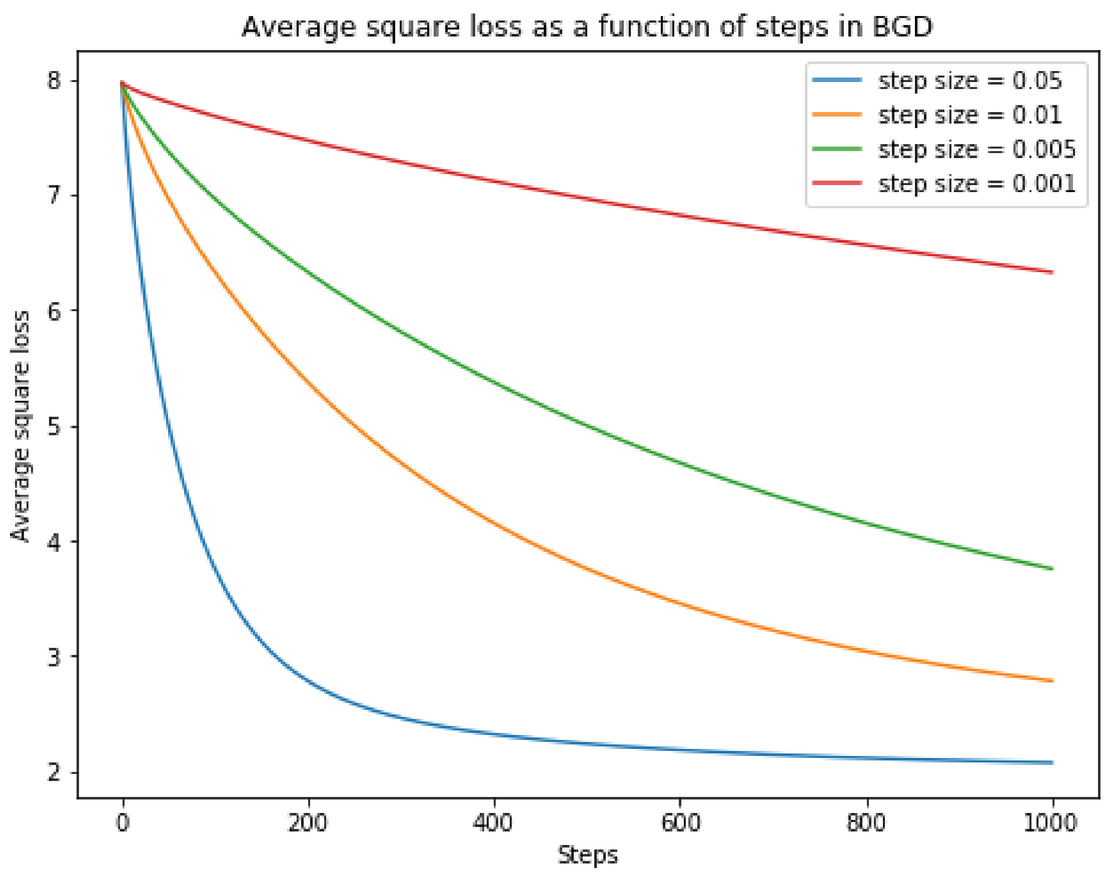
```
[ ]: # Loading the dataset
     print('Loading the dataset')
     df = pd.read_csv('data.txt', delimiter=',')
     X = df.values[:,:-1]
     y = df.values[:,-1]

     print('Split into Training 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)
     X_train_copy = X_train.copy()
     X_test_copy = X_test.copy()

     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 Training and Test
Scaling all to [0, 1]
```

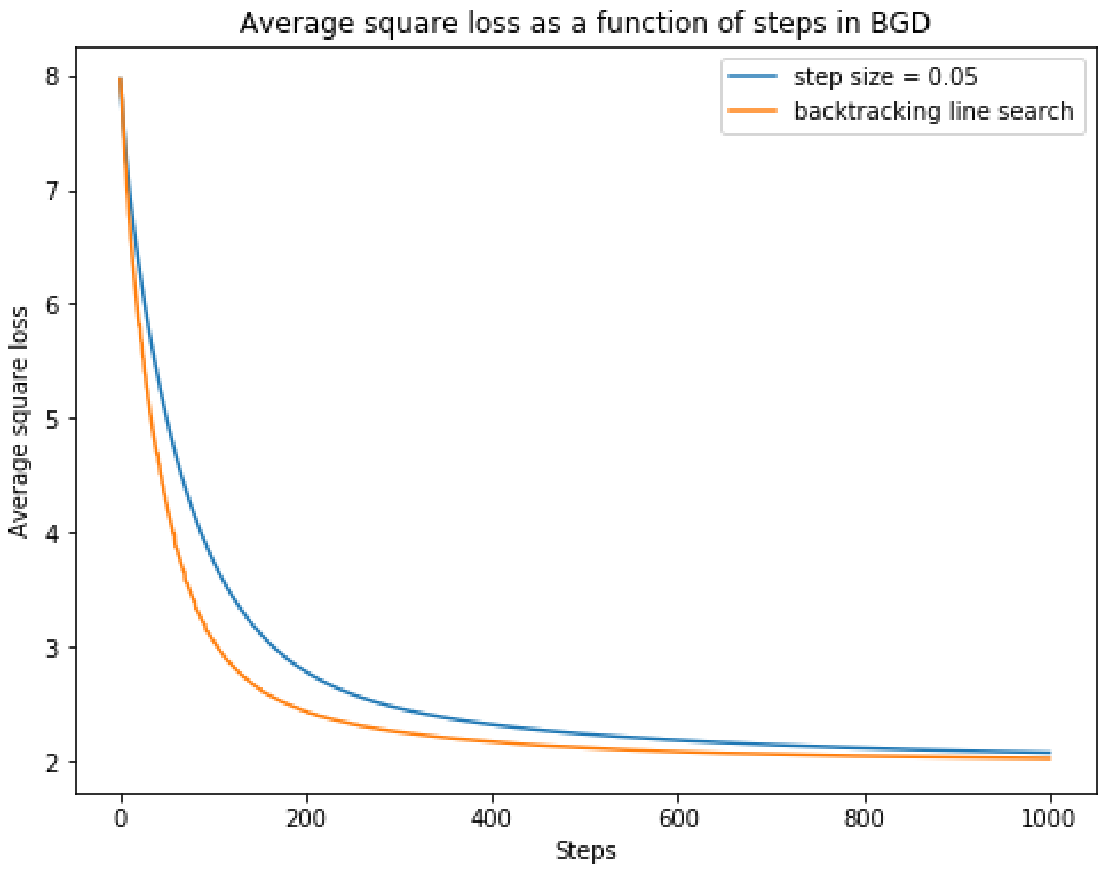
```
[ ]: # Plot
     fig = plt.figure(figsize=(8,6))
     ax = fig.add_subplot(111)
     for a in [0.05, 0.01, 0.005, 0.001]:
         theta_hist1, loss_hist1 = batch_grad_descent(X_train, y_train, alpha=a)
         plt.plot(loss_hist1, label='step size = %r' %a)
     ax.set_xlabel('Steps')
     ax.set_ylabel('Average square loss')
     plt.title('Average square loss as a function of steps in BGD')
     plt.legend(loc='best')
     plt.show()
```



[ ]:
```
#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
```

[ ]:
```
#Plot
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)
theta_hist1, loss_hist1 = batch_grad_descent(X_train, y_train, alpha=0.05)
plt.plot(loss_hist1, label='step size = 0.05')
```

```
theta_hist2, loss_hist2 = batch_grad_descent(X_train, y_train,␣
 ↪alpha='stepsize_search')
plt.plot(loss_hist2, label='backtracking line search')
ax.set_xlabel('Steps')
ax.set_ylabel('Average square loss')
plt.title('Average square loss as a function of steps in BGD')
plt.legend(loc='best')
plt.show()
```



[ ]:
```
def stochastic_grad_descent(X, y, alpha=0.01, num_epoch=1000):
    """
    In this question you will implement stochastic gradient descent
    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.
            num_epoch - number of epochs to go through the whole training set
        Returns:
            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)

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


                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:
```

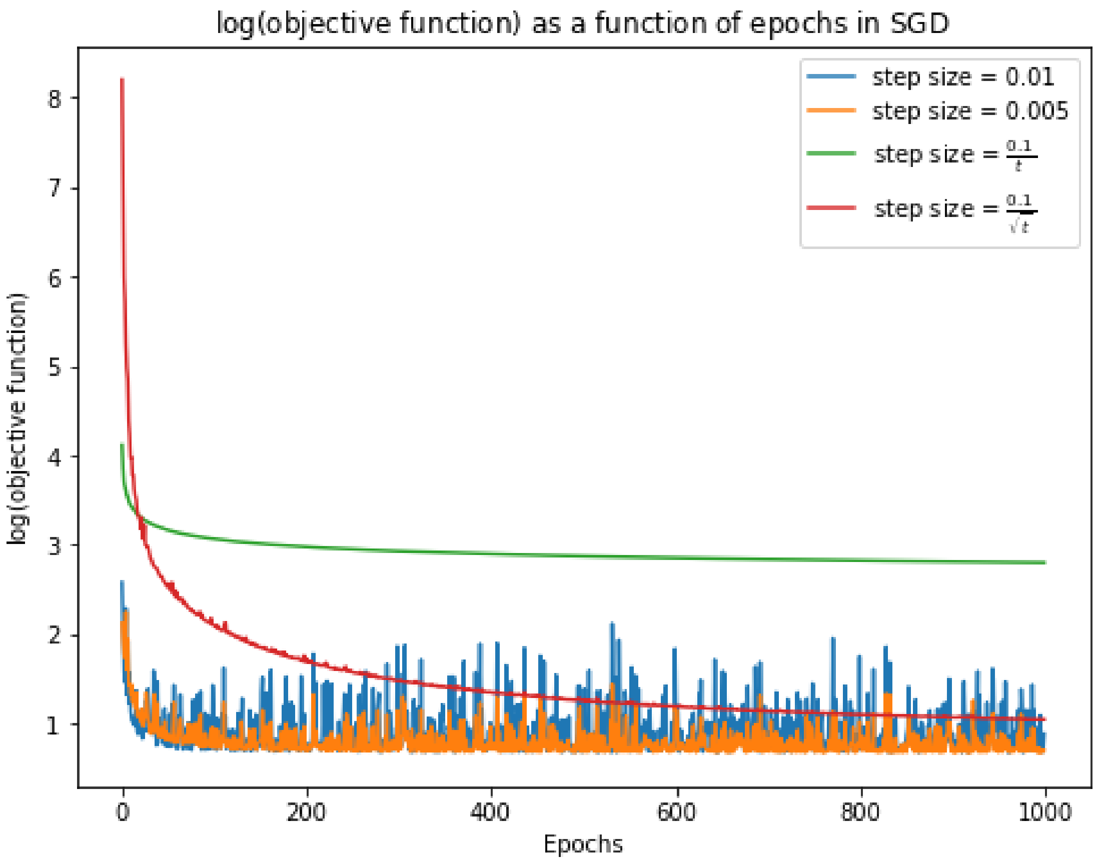
```
                theta = theta - alpha * grad #Update theta

    return theta_hist, loss_hist
```

[ ]:
```
theta_hist1, loss_hist1 = stochastic_grad_descent(X_train, y_train, alpha=0.01)
theta_hist2, loss_hist2 = stochastic_grad_descent(X_train, y_train, alpha=0.005)
theta_hist3, loss_hist3 = stochastic_grad_descent(X_train, y_train, alpha="1/t")
theta_hist4, loss_hist4 = stochastic_grad_descent(X_train, y_train, alpha="1/
 ↪sqrt(t)")

fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111)
plt.plot(np.log(loss_hist1[:,-1]), label='step size = 0.01')
plt.plot(np.log(loss_hist2[:,-1]), label='step size = 0.005')
plt.plot(np.log(loss_hist3[:,-1]), label='step size = $\\frac{0.1}{t}$')
plt.plot(np.log(loss_hist4[:,-1]), label='step size = $\\frac{0.1}{\sqrt{t}}$')

ax.set_xlabel('Epochs')
ax.set_ylabel('$\log$(objective function)')
plt.title('$\log$(objective function) as a function of epochs in SGD')
plt.legend(loc='best')
plt.show()
```



[ ]:
```
```