DS-GA 1003 - Homework 2

Eric Niblock

February 10th, 2021

We have provided you with a file called `ridge_regression_dataset.csv`. Columns `x0` through `x47` correspond to the input and column `y` corresponds to the output. We are trying to fit the data using a linear model and gradient based methods. Please also check the supporting code in `skeleton_code.py`. Throughout this problem, we refer to particular blocks of code to help you step by step.

When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization, features with larger values are treated as "more important", which is not usually desired.

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 validation set or test set. Importantly, 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.

1. **Modify function `feature_normalization` to normalize all the features to $[0, 1]$. Can you use numpy's broadcasting here? Often broadcasting can help to simplify and/or speed up your code. 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.**

   The following code normalizes the features of the training data as described, and provides the same transformation to test data. Before the transformation, constant columns of the training data (and therefore the analogous columns of the test data) are removed.

```
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
    """

    non_cons = ~(train == train[0,:]).all(0)
    train = train[:, non_cons]          ## Removes constant columns
    test = test[:, non_cons]

    mini = np.amin(train,axis=0)        ## Finds column-wise minimums
    train_shift = train - mini          ## Shifts by column minimums

    maxi = np.amax(train_shift,axis=0)  ## Finds column-wise maximums
    train_normalized = train_shift / maxi     ## Divides by column maximums

    test_shift = test - mini
    test_normalized = test_shift / maxi

    return(train_normalized, test_normalized)
```

At the end of the skeleton code, the function `load_data` loads, split into a training and test set, and normalize the data using your `feature_normalization`.
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 $\theta, \boldsymbol{x} \in \mathbb{R}^d$, and we choose θ that minimizes the following "average square loss" objective function:

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

where $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m) \in \mathbb{R}^d \times \mathbb{R}$ 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:

$$h_{\theta,b}(x) = \theta^T \boldsymbol{x} + b,$$

which allows a nonzero intercept term b – sometimes called a "bias" term. The standard way to achieve this, while still maintaining the convenience of the first representation, is to add an extra dimension to $\boldsymbol{x}$ that is always a fixed value, such

as 1, and use $\theta, x \in \mathbb{R}^{d+1}$. Convince yourself that this is equivalent. We will assume this representation.

2. **Let $X \in \mathbb{R}^{m \times (d+1)}$ be the *design matrix*, where the i'th row of X is $\boldsymbol{x}_i$. 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.**

   First, we have that,

$$X = \begin{bmatrix} 1 & x_{1,2} & \cdots & x_{1,d+1} \\ 1 & x_{2,2} & \cdots & x_{2,d+1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{m,2} & \cdots & x_{m,d+1} \end{bmatrix} \quad \boldsymbol{y} = [y_1, y_2, \cdots, y_m]^\top. \tag{1}$$

   Then we can define θ as being,

$$\theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_{d+1} \end{bmatrix} \tag{2}$$

   It follows then that,

$$\begin{aligned} J(\theta) = \frac{1}{m}||X\theta - y||_2^2 &= \frac{1}{m} \left|\left| \begin{bmatrix} 1 & x_{1,2} & \cdots & x_{1,d+1} \\ 1 & x_{2,2} & \cdots & x_{2,d+1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{m,2} & \cdots & x_{m,d+1} \end{bmatrix} \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_{d+1} \end{bmatrix} - \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \right|\right| \\ &= \frac{1}{m} \left|\left| \begin{bmatrix} \theta_1 + \theta_2 x_{1,2} + ... + \theta_{d+1} x_{1,d+1} - y_1 \\ \vdots \\ \theta_1 + \theta_2 x_{m,2} + ... + \theta_{d+1} x_{m,d+1} - y_m \end{bmatrix} \right|\right| \\ &= \frac{1}{m} \left|\left| \begin{bmatrix} h_{\theta,b}(x_1) - y_1 \\ \vdots \\ h_{\theta,b}(x_m) - y_m \end{bmatrix} \right|\right| \\ &= \frac{1}{m} \sum_{i=1}^{m} (h_{\theta,b}(\boldsymbol{x}_i) - y_i)^2 \end{aligned} \tag{3}$$

In other words, in matrix representation, we can express $J(\theta)$ as,

$$J(\theta) = \frac{1}{m}||X\theta - y||_2^2 \tag{4}$$

3. Write down an expression for the gradient of J without using an explicit summation sign.

We have that,

$$\begin{aligned}
\nabla_\theta J(\theta) &= \frac{1}{m}\nabla_\theta ||X\theta - y||_2^2 \\
&= \frac{1}{m}\nabla_\theta \left((X\theta - y)^T(X\theta - y)\right) \\
&= \frac{2X^T \cdot (X\theta - y)}{m}
\end{aligned} \tag{5}$$

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

If we call t the time-step, then we have that,

$$\theta_{t+1} = \theta_t - \eta\nabla_\theta J(\theta) \tag{6}$$

$$\theta_{t+1} = \theta_t - \frac{2\eta X^T \cdot (X\theta_t - y)}{m} \tag{7}$$

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.

The following code computes the square loss using matrix notation,

```
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 = (1/X.shape[0])*np.linalg.norm((X @ theta - y))
    return(loss)
```

6. **Modify the function** `compute_square_loss_gradient`**, to compute $\nabla_\theta J(\theta)$. You may again want to use a small dataset to verify that your** `compute_square_loss_gradient` **function returns the correct value.**

The following code computes the gradient of the square 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)
    """

    grad = (1/X.shape[0])*2*X.T @ (X @ theta - y)
    return(grad)
```

If $J : \mathbb{R}^d \to \mathbb{R}$ is differentiable, then for any vector $h \in \mathbb{R}^d$, the directional derivative of J at θ in the direction h is given by

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

It is also given by the more standard definition of directional derivative,

$$\lim_{\epsilon \to 0} \frac{1}{\epsilon} \left[J(\theta + \epsilon h) - J(\theta) \right] \ .$$

The former form gives a better approximation to the derivative when we are using small (but not infinitesimally small) ϵ. We can approximate this directional derivative by choosing a small value of $\epsilon > 0$ and evaluating the quotient above. We can get an approximation to the gradient by approximating the directional derivatives in each coordinate direction and putting them together into a vector. In other words, take $h = (1, 0, 0, \ldots, 0)$ to get the first component of the gradient. Then take $h = (0, 1, 0, \ldots, 0)$ to get the second component, and so on.

7. **Complete the function `grad_checker` according to the documentation of the function given in the `skeleton_code.py`. Alternatively, you may complete the function `generic_grad_checker` which can work for any objective function.**

Below is the code which numerically checks the gradient,

```python
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

    for d in range(num_features):
        h = np.zeros(num_features)
        h[d] = 1
        higher = compute_square_loss(X, y, theta+(epsilon*h))
        lower = compute_square_loss(X, y, theta-(epsilon*h))
        approx_grad[d] = (higher-lower)/(2*epsilon)

    norm = np.linalg.norm(true_gradient - approx_grad)

    if norm < tolerance:
        return(True)
    else:
        return(False)
```

You should be able to check that the gradients you computed above remain correct throughout the learning below.

We can confirm that this code works. Take the following example, which returns True,

```
X = np.array([[1,8,4,2,56],[1,4,2,6,96],[3,2,1,6,6]])
y = np.array([4,3.3,2])
theta = np.array([4,1,6,4,3])

grad_checker(X,y,theta, epsilon=0.01, tolerance=1e-4)
```

We will now finish the job of running regression on the training set.

8. **Complete** `batch_gradient_descent`**. Note the phrase *batch* gradient descent distinguishes between *stochastic* gradient descent or more generally *minibatch* gradient descent.**

The following code is used to employ full-batch gradient descent, where the gradient check is used to stop the descent if the algorithm begins to diverge uncontrollably,

```
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
```

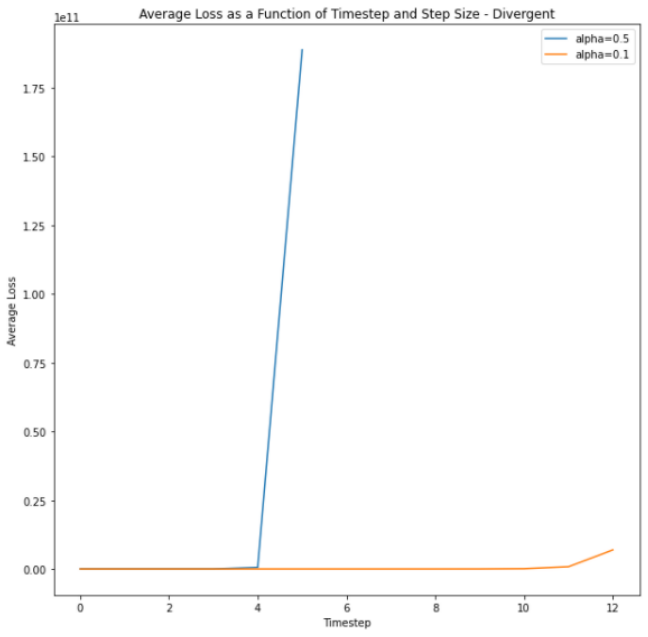
```
    minimize the average square loss objective.

    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

    for n in range(num_step+1):

        loss_hist[n] = compute_square_loss(X, y, theta)
        theta_hist[n,:] = theta
        grad = compute_square_loss_gradient(X, y, theta)

        if grad_check == True:
            if grad_checker(X,y,theta, epsilon=0.01, tolerance=1e-4) == False:
                print('Error: Gradient Incorrect, Stopped at Timestep: ',n)
                break

        theta_temp = theta - alpha*grad
        theta = theta_temp

    return(theta_hist,loss_hist)
```

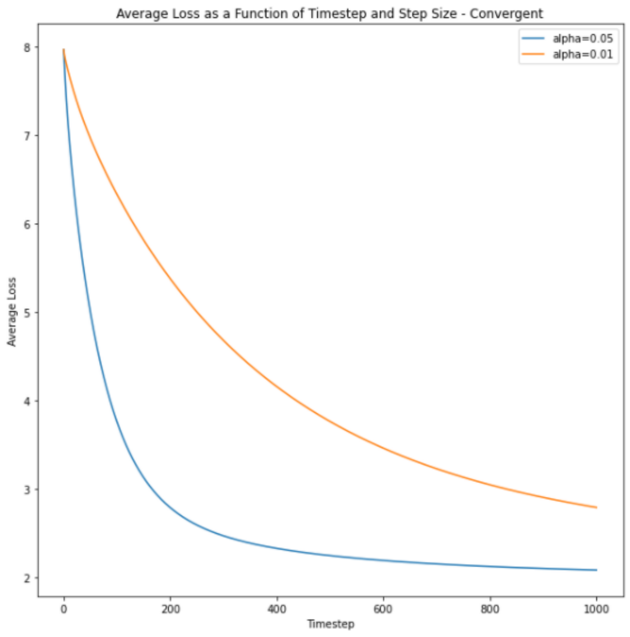
9. **Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge. Starting with a step-size of 0.1, try various different fixed step sizes to see which converges most quickly and/or which diverge. As a minimum, try step sizes 0.5, 0.1, .05, and .01. Plot the average square loss on the training set as a function of the number of steps for each step size. Briefly summarize your findings.**

First, we supply the code necessary to evaluate the various step-sizes,

```
X_train, y_train, X_test, y_test = load_data()
theta_hist0, loss_hist0 = batch_grad_descent(X_train, y_train, alpha=0.5,
        num_step=1000, grad_check=True)
theta_hist1, loss_hist1 = batch_grad_descent(X_train, y_train, alpha=0.1,
        num_step=1000, grad_check=True)
theta_hist2, loss_hist2 = batch_grad_descent(X_train, y_train, alpha=0.05,
        num_step=1000, grad_check=True)
theta_hist3, loss_hist3 = batch_grad_descent(X_train, y_train, alpha=0.01,
        num_step=1000, grad_check=True)
```

We generated two plots which reveal the average training loss as a function of the step-size and time-step. The first plot reveals divergent trends, resulting from a step-size which is too large. The second plot reveals convergent plots, with step-sizes that are sufficiently small,





It is clear from the plots that a step size of $\alpha = 0.05$ is ideal when compared to the other step-sizes. Step-sizes which are larger than or equal to 0.1 diverge, and thus cause the gradient descent to halt before reaching 1000 time-steps. In these cases the step-size is too large, and the update method continually misses the minimum, jumping farther and farther away.

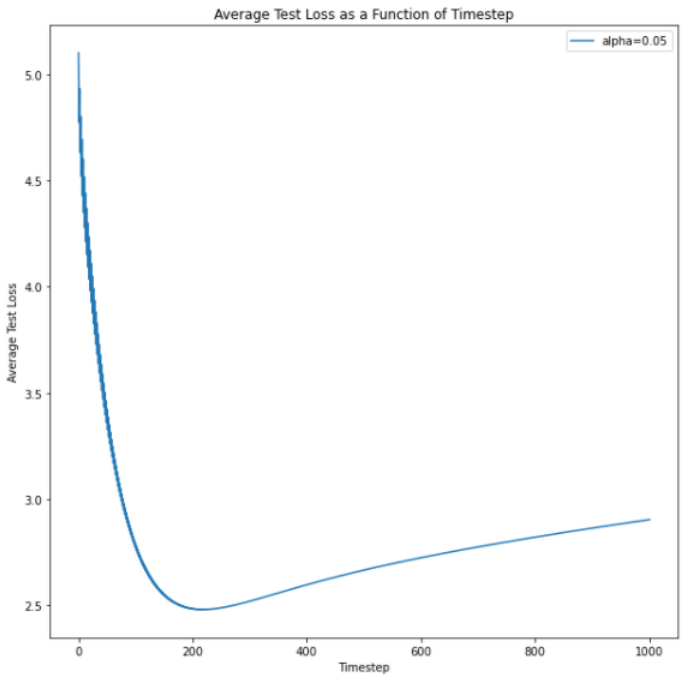
10. For the learning rate you selected above, plot the average test loss as a function of the iterations. You should observe overfitting: the test error first decreases

and then increases.

The code necessary to evaluate the test loss is provided here,

```
num_step = 1000
test_loss_hist = np.zeros(num_step + 1)
for t in range(len(theta_hist)):
    test_loss_hist[t] = compute_square_loss(X_test, y_test, theta_hist[t])
```

Here, we see the average test loss as a function of time-step at a learning rate of $\alpha = 0.05$.



As expected, the test loss first decreases to a minimum around $t = 200$ and then begins to increase again due to over-fitting.

We will add ℓ_2 regularization to linear regression. 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

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

where λ is the regularization parameter, which controls the degree of regularization. Note that the bias term (which we included as an extra dimension in θ) is being regularized as well as the other parameters. Sometimes it is preferable to treat this term separately.

11. **Compute the gradient of $J_\lambda(\theta)$ and write down the expression for updating θ in the gradient descent algorithm. (Matrix/vector expression, without explicit summation)**

We have that $J_\lambda(\theta)$ is in part composed of $J(\theta)$, such that,

$$J_\lambda(\theta) = J(\theta) + \lambda\theta^T\theta \tag{8}$$

And therefore the gradient is much easier to calculate,

$$\begin{aligned}\nabla_\theta J_\lambda(\theta) &= \frac{2X^T \cdot (X\theta - y)}{m} + \nabla_\theta(\lambda\theta^T\theta) \\ &= \frac{2X^T \cdot (X\theta - y)}{m} + 2\lambda\theta\end{aligned} \tag{9}$$

12. **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)
"""

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

13. 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_features) for instance, theta in step 0 should
    be theta_hist[0], theta in step(num_step+1) is theta_hist[-1]
    loss hist - the history of average square loss function without the
    regularization term, 1D numpy array.
"""

num_instances, num_features = X.shape[0], X.shape[1]
theta = np.zeros(num_features) #Initialize theta
theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
loss_hist = np.zeros(num_step+1) #Initialize loss_hist

for n in range(num_step+1):

    loss_hist[n] = compute_square_loss(X, y, theta)
    theta_hist[n,:] = theta
    grad = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)


    if grad_checker(X,y,theta, epsilon=0.01, tolerance=1e-4) == False:
        print('Error: Gradient Incorrect, Stopped at Timestep: ',n)
        break

    theta_temp = theta - alpha*grad
    theta = theta_temp

return(theta_hist,loss_hist)
```

Our goal is to find λ that gives the minimum average square loss on the test set. So you should start your search very broadly, looking over several orders of magnitude. For example, $\lambda \in \{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\}$. Then you can zoom in on the best range. Follow the steps below to proceed.

14. **Choosing a reasonable step-size, plot 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 the training iterations for various values of λ. What do you notice in terms of overfitting?**

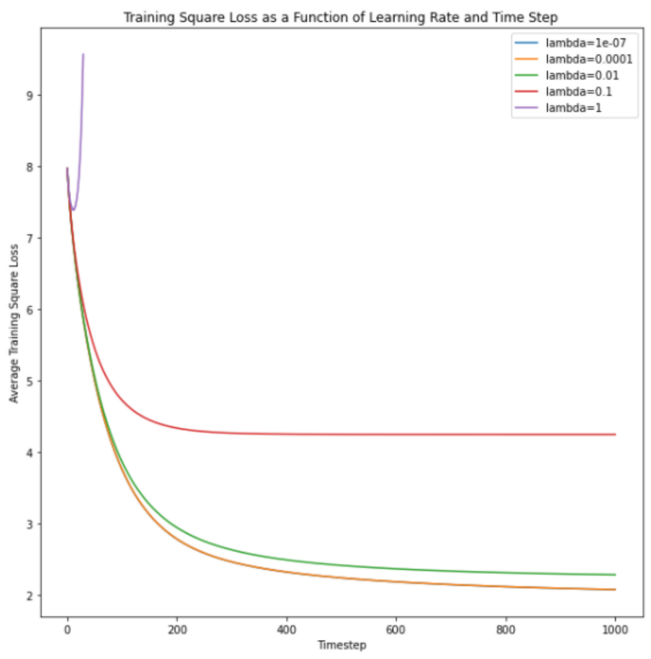
Here is the code employed to provide the training losses,

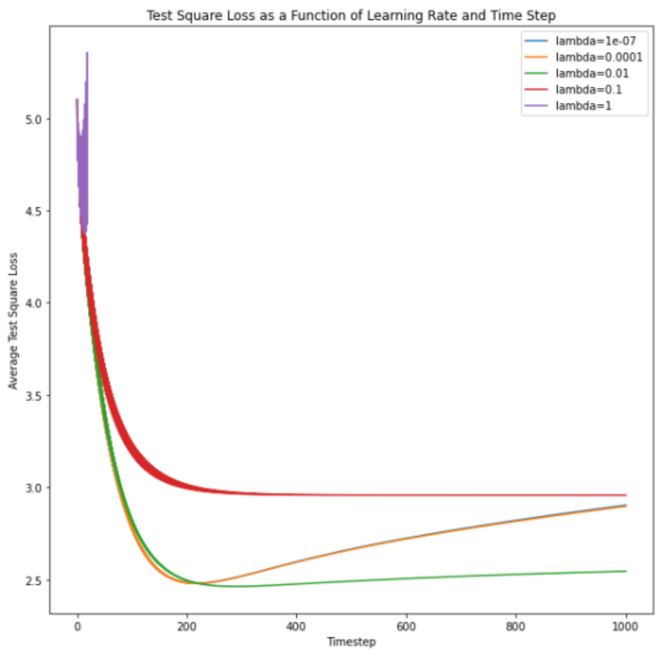
```
losses = []
thetas = []
for lambda_reg_s in [10**-7,10**-4,0.01,0.1,1]:
    theta, loss = regularized_grad_descent(X_train, y_train, alpha=0.05,
            lambda_reg=lambda_reg_s, num_step=1000)
    losses.append(loss)
    thetas.append(theta)
```

And the test losses,

```
test_losses = []
for choice in thetas:
    num_step = 1000
    test_loss_hist = np.zeros(num_step + 1)
    for t in range(len(choice)):
        test_loss_hist[t] = compute_square_loss(X_test, y_test, choice[t])
    test_losses.append(test_loss_hist)
```

Below we have plots for the test and training losses for various values of λ,





Note that the plot for $\lambda = 1$ diverges off of our scale. Additionally, the loss associated with $\lambda = 10^{-7}$ is nearly identical to the loss regarding $\lambda = 10^{-4}$. Therefore, it isn't all that visible on either plot. In terms of overfitting, notice that values of $\lambda < 1$ overfit, as seen by the testing losses which decrease to minimums around $t = 200$ and then begin to increase. You may also notice that the test loss plots are thick in some regions. This is because the test loss is actually oscillating around some minimal region, though overall is decreasing.

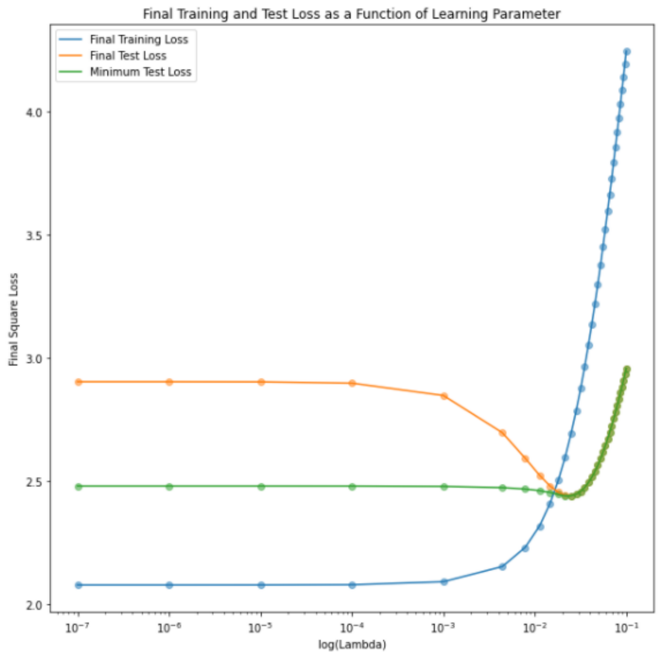
15. Plot the training average square loss and the test average square loss at the end of training as a function of λ. You may want to have $\log(\lambda)$ on the x-axis rather than λ. Which value of λ would you choose?

The code used to generate our square loss as a function of λ as the end of training and test, as well as at the minimum test loss value is provided,

```
losses = []
thetas = []
values = np.append([10**-7,10**-6,10**-5,10**-4,],np.linspace(10**-3,0.1,30))
for lambda_reg_s in values:
    theta, loss = regularized_grad_descent(X_train, y_train, alpha=0.05,
            lambda_reg=lambda_reg_s, num_step=1000)
    losses.append(loss)
    thetas.append(theta)
test_loss = []
```

```
for theta in thetas:
    test_loss.append(compute_square_loss(X_test, y_test, theta[-1]))
min_loss = []
for theta in thetas:
    temp = []
    for t in theta:
        temp.append(compute_square_loss(X_test, y_test, t))
    min_loss.append(min(temp))
```

We then provided a plot of the final training loss, final test loss, and minimum test loss, each as a function of $log(\lambda)$,



From analyzing the final testing loss, it is clear that we should select $\lambda \approx 0.025$.

16. Another heuristic of regularization is to *early-stop* the training when the test error reaches a minimum. Add to the last plot the minimum of the test average square loss along training as a function of λ. Is the value λ you would select with early stopping the same as before?

First, observe that the plot from the previous question includes the minimum test loss. The value of λ that creates the minimum testing loss is the same when considering either the final testing loss, or the minimum testing loss as a result of early-stopping. This value of λ is approximately 0.025. We would therefore select the same value of λ as before.

17. What θ would you select in practice and why?

We should select the value of θ which minimizes the testing loss, whether at the end of training or through early-stopping (in this case, it doesn't matter). We note that this value of θ occurs when $\alpha = 0.05$ and $\lambda = 0.025$. The value of theta is not given here because of its size.

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.

In SGD, rather than taking $-\nabla_\theta J(\theta)$ as our step direction to minimize

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

we take $-\nabla_\theta f_i(\theta)$ 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. 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.

18. Show that the objective function

$$J_\lambda(\theta) = \frac{1}{m}\sum_{i=1}^{m} \left(h_\theta(\boldsymbol{x}_i) - y_i\right)^2 + \lambda\theta^T\theta$$

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

Since we have that,

$$J_\lambda(\theta) = \frac{1}{m}\sum_{i=1}^{m} \left(h_\theta(\boldsymbol{x}_i) - y_i\right)^2 + \lambda\theta^T\theta = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta) \tag{10}$$

It is trivial that,

$$f_i(\theta) = (h_\theta(\boldsymbol{x}_i) - y_i)^2 + \lambda\theta^T\theta \tag{11}$$
$$f_i(\theta) = \left(\theta^T\boldsymbol{x_i} - y_i\right)^2 + \lambda\theta^T\theta \tag{12}$$

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

We have that,

$$\begin{aligned} E[\nabla f_i(\theta)] &= E\left[2(\theta^T x_i - y_i)\cdot x_i + 2\lambda\theta\right] \\ &= 2E[(\theta^T x_i)\cdot x_i] - 2E[y_i x_i] + 2E[\lambda\theta] \\ &= 2E\left[\left(\sum_{j=1}^{d}\theta_d x_{i,d}\right)\cdot x_i\right] - 2E[y_i x_i] + 2\lambda\theta \end{aligned} \tag{13}$$

Now, since each x_i is being chosen uniformly at random from the possible x_is, the probability of any x_i being chosen is simply $\frac{1}{m}$. Now, we can evaluate the expectations,

$$E[\nabla f_i(\theta)] = 2\sum_{i=1}^{m}\frac{1}{m}\left(\left(\sum_{j=1}^{d}\theta_d x_{i,d}\right)\cdot x_i\right) - 2\sum_{i=1}^{m}\frac{y_i x_i}{m} + 2\lambda\theta \tag{14}$$

However, we can remove the summations by writing this in matrix notation, yielding,

$$E[\nabla f_i(\theta)] = \frac{2}{m}\left(X^T X\theta - X^T y\right) + 2\lambda\theta \tag{15}$$

However, this is precisely $\nabla_\theta J_\lambda(\theta)$. Therefore $\nabla f_i(\theta)$ is an unbiased estimator of $\nabla_\theta J_\lambda(\theta)$.

20. Write down the update rule for θ in SGD for the ridge regression objective function.

If we call t the time-step, then we have,

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta f_i(\theta) \tag{16}$$

Which is given by,

$$\theta_{t+1} = \theta_t - 2\eta((\theta^T x_i - y_i) \cdot x_i + \lambda\theta) \tag{17}$$

21. Implement `stochastic_grad_descent`.

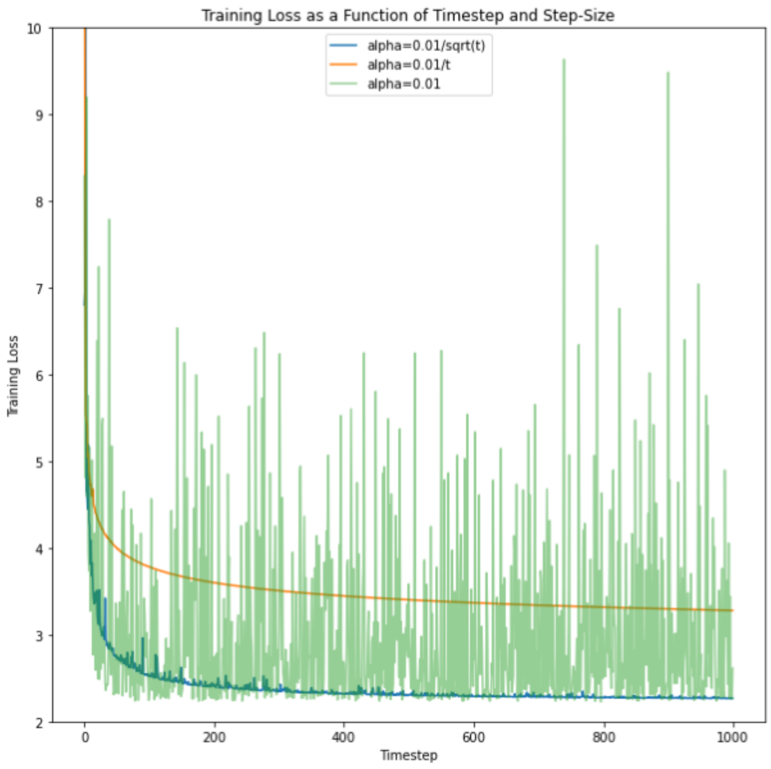
```python
def stochastic_grad_descent(X, y, alpha=0.1, lambda_reg=10**-2,
                            num_epoch=1000, eta0=False):
    """
    In this question you will implement stochastic gradient descent with
    regularization term

    Args:
        X - the feature vector, 2D numpy array of size(num_instances, num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        alpha - string or float, step size in gradient descent
                NOTE: In SGD, it's not a good idea to use a fixed step size.
                Usually it's set to 1/sqrt(t) or 1/t
                if alpha is a float, then the step size in every step is the float.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t).
                if alpha == "1/t", alpha = 1/t.
        lambda_reg - the regularization coefficient
        num_epoch - number of epochs to go through the whole training set

    Returns:
        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))
    loss_hist = np.zeros((num_epoch, num_instances))

    rc = list(range(num_instances))


    for n in range(num_epoch):

        np.random.shuffle(rc)

        if alpha == "1/sqrt(t)":
            step = 0.01/((n+1)**0.5)
        if alpha == "1/t":
            step = 0.01/(n+1)
        if type(alpha) == float:
            step = alpha

        for r in rc:
            xi = X[r]
            xi = xi[np.newaxis,:]
            yi = y[r]
            loss_hist[n,r] = compute_square_loss(X, y, theta)
            theta_hist[n,r,:] = theta

            grad = compute_regularized_square_loss_gradient(xi, yi, theta, lambda_reg)

            theta_temp = theta - step*grad
            theta = theta_temp

    return(theta_hist,loss_hist)
```

22. Use SGD to find θ^*_λ that minimizes the ridge regression objective for the λ you selected in the previous problem. (If you could not solve the previous problem, choose $\lambda = 10^{-2}$). 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 are 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). How do the results compare?

We ran stochastic gradient descent with different step-sizes using the code below,

```
thetas, loss_sqrt = stochastic_grad_descent(X_train, y_train,
        alpha="1/sqrt(t)", lambda_reg=10**-2, num_epoch=1000, eta0=False)
thetas, loss_1t = stochastic_grad_descent(X_train, y_train,
        alpha="1/t", lambda_reg=10**-2, num_epoch=1000, eta0=False)
thetas, loss = stochastic_grad_descent(X_train, y_train,
        alpha=0.01, lambda_reg=10**-2, num_epoch=1000, eta0=False)
```

We then plotted the training loss at the end of each epoch for each step-size method, achieving the following,



First, it is important to note that it was necessary to use $C = 0.01$ to achieve any type of convergence using any method. Furthermore, we see that the constant step-size results in a non-convergent training loss as a function of epoch. This is expected. When using the decreasing step-sizes, we do achieve convergence, with $\alpha = \frac{1}{\sqrt{t}}$ providing a final training loss that is comparable with the other gradient descent methods.

A few remarks about the question above:

- **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.**

- **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 statistical learning theory terminology, 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 close enough to the minimizer.**

In this second problem set we will examine a classification problem. To do so we will use the MNIST dataset[1] which is one of the traditional image benchmark for machine learning algorithms. We will only load the data from the 0 and 1 class, and try to predict the class from the image. You will find the support code for this problem in `mnist_classification_source_code.py`. Before starting, take a little time to inspect the data. Load `X_train, y_train, X_test, y_test` with `pre_process_mnist_01()`. Using the function `plt.imshow` from `matplotlib` visualize some data points from `X_train` by reshaping the 764 dimensional vectors into 28×28 arrays. Note how the class labels '0' and '1' have been encoded in `y_train`. No need to report these steps in your submission.

We will use here again a linear model, meaning that we will fit an affine function,

$$h_{\theta,b}(\boldsymbol{x}) = \theta^T \boldsymbol{x} + b,$$

with $\boldsymbol{x} \in \mathbb{R}^{764}$, $\boldsymbol{\theta} \in \mathbb{R}^{764}$ and $b \in \mathbb{R}$. This time we will use the logistic loss instead of the squared loss. Instead of coding everything from scratch, we will also use the package `scikit learn` and study the effects of ℓ_1 regularization. You may want to check that you have a version of the package up to date (0.24.1).

23. **Recall the definition of the logistic loss between target y and a prediction $h_{\theta,b}(\boldsymbol{x})$ as a function of the margin $m = yh_{\theta,b}(\boldsymbol{x})$. Show that given that we chose the convention $y_i \in \{-1, 1\}$, our objective function over the training data $\{\boldsymbol{x}_i, y_i\}_{i=1}^m$ can be re-written as**

$$L(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (1 + y_i) \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y_i) \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)}).$$

We have that,

[1] `http://yann.lecun.com/exdb/mnist/`

$$\ell_{Logistic}(\theta) = \log(1 + e^{-m}) = \log(1 + e^{-y_i h_{\theta,b}(\boldsymbol{x}_i)}) \tag{18}$$

Which implies that,

$$\ell_{Logistic}(\theta) = \begin{cases} \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}), & \text{if } y_i = 1 \\ \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)}), & \text{if } y_i = -1 \end{cases} \tag{19}$$

Now let's define a different function $\ell(\theta)$,

$$\ell(\theta) = \frac{1}{2}\Big((1 + y_i)\log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y_i)\log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)})\Big) \tag{20}$$

Notice that this function provides the same implication as before,

$$\ell(\theta) = \begin{cases} \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}), & \text{if } y_i = 1 \\ \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)}), & \text{if } y_i = -1 \end{cases} \tag{21}$$

So, therefore, we have,

$$L(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(1+y_i)\log(1+e^{-h_{\theta,b}(\boldsymbol{x}_i)})+(1-y_i)\log(1+e^{h_{\theta,b}(\boldsymbol{x}_i)}) = \sum_{i=1}^{m}\ell_{Logistic,i}(\theta) \tag{22}$$

It is clear that this is the objective function, because it is the sum of logistic losses.

24. What will become the loss function if we regularize the coefficients of θ with an ℓ_1 penalty using a regularization parameter α ?

We can add ℓ_1 regularization with parameter α as below,

$$L(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(1+y_i)\log(1+e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1-y_i)\log(1+e^{h_{\theta,b}(\boldsymbol{x}_i)}) + \alpha\sum_{j=1}^{d}|\theta_j| \quad (23)$$

Where d is the size of the vector θ.

We are going to use the module `SGDClassifier` from scikit learn. In the code provided we have set a little example of its usage. By checking the online documentation[2], make sure you understand the meaning of all the keyword arguments that were specified. We will keep the learning rate schedule and the maximum number of iterations fixed to the given values for all the problem. Note that scikit learn is actually implementing a fancy version of SGD to deal with the ℓ_1 penalty which is not differentiable everywhere, but we will not enter these details here.

25. To evaluate the quality of our model we will use the classification error, which corresponds to the fraction of incorrectly labeled examples. For a given sample, the classification error is 1 if no example was labeled correctly and 0 if all examples were perfectly labeled. Using the method `clf.predict()` from the classifier write a function that takes as input an `SGDClassifier` which we will call `clf`, a design matrix `X` and a target vector `y` and returns the classification error. You should check that your function returns the same value as `1 - clf.score(X, y)`.

The following function returns the rate of classification error,

```
def classification_error(clf, X, y):
      preds = clf.predict(X)
      correct = np.count_nonzero(preds == y)
      return(1 - (correct/len(y)))
```

To speed up computations we will subsample the data. Using the function `sub_sample`, restrict `X_train` and `y_train` to `N_train = 100`.

26. Report the test classification error achieved by the logistic regression as a function of the regularization parameters α (taking 10 values between 10^{-4} and 10^{-1}). You should make a plot with α as the x-axis in log scale. For each value of α, you should repeat the experiment 10 times so has to finally report the mean value and the standard deviation. You should use `plt.errorbar` to plot

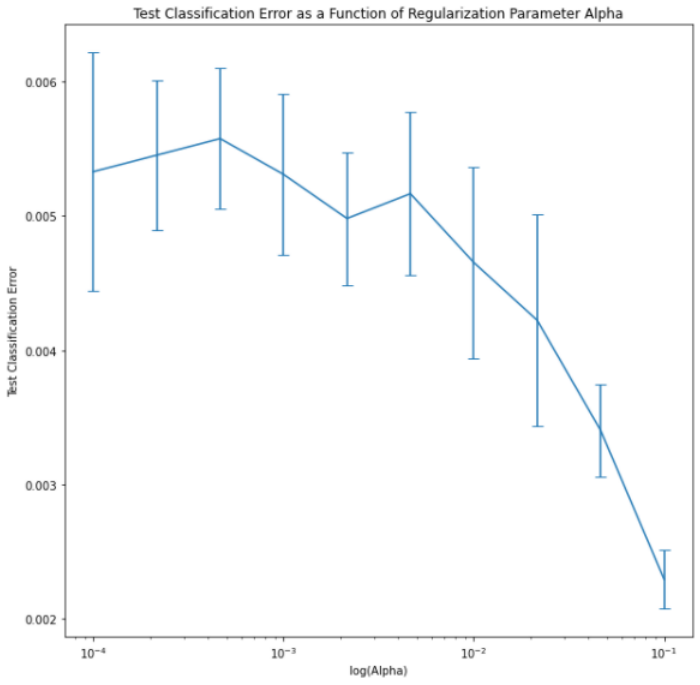
[2]https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

the standard deviation as error bars.

The following is the code used to generate the plot of classification error as a function of alpha,

```
alphas = np.logspace(-4,-1,10)
alpha_means = []
alpha_stds = []

for a in alphas:
    errs = []
    for t in range(10):
        X_sub, y_sub = sub_sample(100, X_train, y_train)
        clf = SGDClassifier(loss='log', max_iter=1000,
                                    tol=1e-3,
                                    penalty='l1', alpha=a,
                                    learning_rate='invscaling',
                                    power_t=0.5,
                                    eta0=0.01,
                                    verbose=1)
        clf.fit(X_sub, y_sub)
        err = class_error(clf, X_test, y_test)
        errs.append(err)
    alpha_means.append(np.mean(errs))
    alpha_stds.append(np.std(errs))
```

These are the results, which suggest that increasing α (until some threshold) decreases the classification error. In other words, adding regularization is effective in lowering classification error.



27. Which source(s) of randomness are we averaging over by repeating the experiment?

By repeating the experiment multiple times, and selecting a new subset of the training data each run, we are averaging over the randomness that occurs when selecting a random sample. It is always possible that sample selection will result in sampling error. This occurs when our sample is a biased representation of our population. By averaging over different samples of the training data, we remove the randomness associated with selecting a random sample, which helps to eradicate bias.

28. What is the optimal value of the parameter α among the values you tested?

Of the values of α that we tested, $\alpha = 0.1$ is optimal because it reduces the classification error to a minimum within our domain (in other words, there may be a more ideal α, though of the ones we tested, $\alpha = 0.1$ provides the best results).

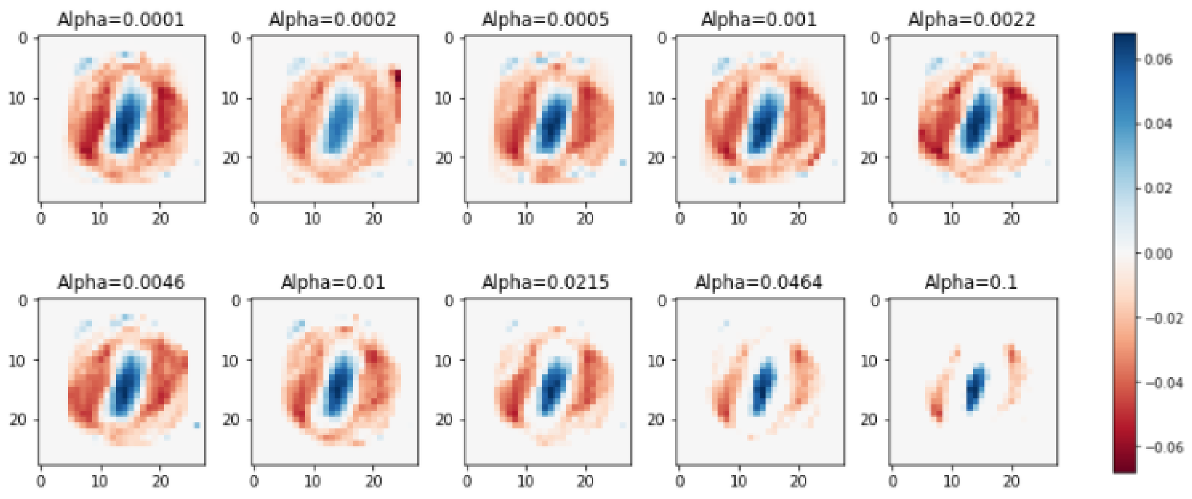
29. Finally, for one run of the fit for each value of α plot the value of the fitted θ. You can access it via `clf.coef_`, and should reshape the 764 dimensional vector to a 28×28 arrray to visualize it with `plt.imshow`. Defining

scale = `np.abs(clf.coef_).max()`, you can use the following keyword arguments (`cmap=plt.cm.RdBu, vmax=scale, vmin=-scale`) which will set the colors nicely in the plot. You should also use a `plt.colorbar()` to visualize the values associated with the colors.

Below is the code used to generate the data necessary for our visualization,

```
alphas = np.logspace(-4,-1,10)
thetas = []

for a in alphas:
    X_sub, y_sub = sub_sample(100, X_train, y_train)
    clf = SGDClassifier(loss='log', max_iter=1000,
                        tol=1e-3,
                        penalty='l1', alpha=a,
                        learning_rate='invscaling',
                        power_t=0.5,
                        eta0=0.01,
                        verbose=1)
    clf.fit(X_sub, y_sub)
    thetas.append(clf.coef_)

data = []
for t in thetas:
    re_t = np.reshape(t,(28,28))
    scale = np.abs(re_t).max()
    data.append((re_t,scale))
```

And here is the resulting visualization,



30. What can you note about the pattern in θ? What can you note about the effect of the regularization?

The resulting pattern regarding θ and effect of regularization are fairly obvious. As we progress to larger values of α, regularization becomes more prominent, and many of the coefficients of θ are being set to zero. This is the expected result when using ℓ_1 regularization. As α increases, regularization contributes more to the overall loss, and in order to be reduced, the algorithm removes emphasis from a plethora of features.