Homework 2_DS-GA 1003

February 22, 2021

Name: Zhuoyuan Xu (Kallen Xu)

NetID: zx1137

Due Date: Feb 22, 2021

Gradient Descent for Ridge(less) Linear Regression

1 Feature Normalization

Problem 1

[1]:
```
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import numpy.linalg as LA
import copy
```

[2]:
```
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
    """
    # The first way to implement this function: looping
    train_min = []
    train_max = []
    train_normalized = train
    test_normalized = test
```

```
    kept_col = []
    for idx in range(len(train[0])):
        train_min.append(train[:, idx].min())
        train_max.append(train[:, idx].max())
        if abs(train_max[idx]-train_min[idx]) > 1e-6:
            train_normalized[:, idx] = (train_normalized[:,␣
↪idx]-train_min[idx])/(train_max[idx]-train_min[idx])
            test_normalized[:, idx] = (test_normalized[:, idx]-train_min[idx])/
↪(train_max[idx]-train_min[idx])
            kept_col.append(idx)
    train_normalized = train_normalized[:, kept_col]
    test_normalized = test_normalized[:, kept_col]


    # The second way to implement this function: broadcasting
    #train_min = train_normalized.min(axis = 0)
    #train_max = train_normalized.max(axis = 0)
    #train_normalized = np.delete(train, np.all(train_min == train_max), 1)
    #train_normalized = np.delete(test, np.all(train_min == train_max), 1)
    #train_normalized = (train_normalized - train_min) / (train_max - train_min)
    #test_normalized = (test_normalized - train_min)/(train_max - train_min)

    return train_normalized, test_normalized
```

[3]:
```
def load_data():
    #Loading the dataset
    print('loading the dataset')

    df = pd.read_csv('ridge_regression_dataset.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)
    X_train = np.hstack((X_train, np.ones((X_train.shape[0], 1))))  # Add bias␣
↪term
    X_test = np.hstack((X_test, np.ones((X_test.shape[0], 1))))
    return X_train, y_train, X_test, y_test
```

[4]:
```
X_train, y_train, X_test, y_test = load_data()
print(X_train)
```

```
loading the dataset
Split into Train and Test
Scaling all to [0, 1]
[[1.         1.         1.         … 0.13241261 0.13241261 1.        ]
 [1.         1.         1.         … 0.92975578 0.92975578 1.        ]
 [1.         1.         1.         … 0.91055382 0.91055382 1.        ]
 …
 [1.         0.         0.         … 0.04886354 0.04886354 1.        ]
 [1.         1.         1.         … 0.58441163 0.58441163 1.        ]
 [0.         0.         0.         … 0.03210449 0.03210449 1.        ]]
```

2 Linear Regression

Problem 2

The objective function $J(\theta)$ in matrix/vector expression can be written as

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

Problem 3

The gradient of J can be written as

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

Problem 4

The expression for updating θ in the gradient descent algorithm for a step size η is

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

Problem 5

[5]:
```
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
    """
    m = len(X)
```

```
    #loss = (X @ np.transpose(theta) - y.T) @ np.transpose((X @ np.
↪transpose(theta) - y.T))/m
    loss = LA.norm(X@theta.T - y.T)**2/m
    return loss
```

Problem 6

[6]:
```
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)
    """
    m = len(X)
    grad = 2 * (X.T @ X @ theta.T - X.T @ y.T) / m
    return grad
```

3 Gradient Checker

Problem 7

[7]:
```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
    e = [0] * num_features
    for i in range(num_features):
        e_i = copy.deepcopy(e)
        e_i[i] = 1
        approx_grad[i] = (compute_square_loss(X, y, theta+epsilon*np.
↪array(e_i)) - compute_square_loss(X, y, theta-epsilon*np.array(e_i)))/
↪(2*epsilon)
    dist = LA.norm(approx_grad - true_gradient)
    return dist <= tolerance
```

4 Batch Gradient Descent

Problem 8

[8]:
```
def batch_grad_descent(X, y, alpha, num_step, grad_check):
    """
    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
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta)

    for s in range(1, num_step+1):
        theta_hist[s] = theta_hist[s-1] - alpha *␣
↪compute_square_loss_gradient(X, y, theta_hist[s-1])
        loss_hist[s] = compute_square_loss(X, y, theta_hist[s])
        if grad_check:
            if grad_checker(X, y, theta_hist[s]) == False:
                print('Overflow step size', alpha, 'and num step', num_step)
                break

    return theta_hist, loss_hist
```

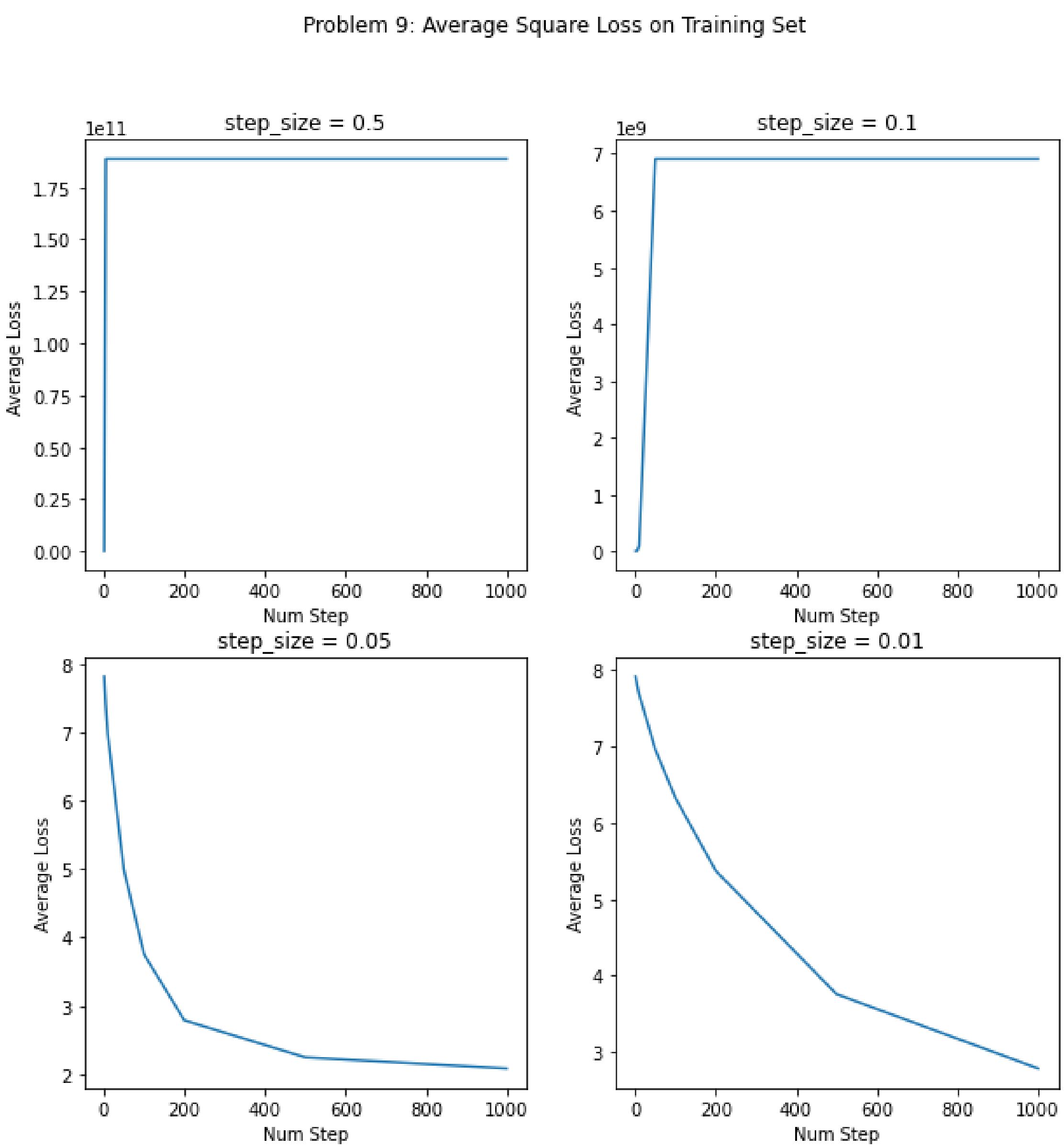
Problem 9

[10]:
```
fig, ax = plt.subplots(2, 2, figsize=(10, 10))
steps = [0.5, 0.1, 0.05, 0.01]
numstep = [1, 5, 10, 50, 100, 200, 500, 1000]
i = 0
for s in steps:
    loss = []
    for n in numstep:
        theta_hist, loss_hist = batch_grad_descent(X_train, y_train, s, n, True)
        temp = [e for e in loss_hist if e!= 0][-1]
        loss.append(temp)
    ax[i//2, i%2].plot(numstep, loss)
    ax[i//2, i%2].set_xlabel('Num Step')
    ax[i//2, i%2].set_ylabel('Average Loss')
    ax[i//2, i%2].set_title('step_size = {}'.format(s))
    i += 1
fig.suptitle('Problem 9: Average Square Loss on Training Set')
```

```
Overflow step size 0.5 and num step 5
Overflow step size 0.5 and num step 10
Overflow step size 0.5 and num step 50
Overflow step size 0.5 and num step 100
```

```
Overflow step size 0.5 and num step 200
Overflow step size 0.5 and num step 500
Overflow step size 0.5 and num step 1000
Overflow step size 0.1 and num step 50
Overflow step size 0.1 and num step 100
Overflow step size 0.1 and num step 200
Overflow step size 0.1 and num step 500
Overflow step size 0.1 and num step 1000
```

[10]: Text(0.5, 0.98, 'Problem 9: Average Square Loss on Training Set')



Note in the first 2 plots, because the grad_checker notifies the function to stop updating the theta and loss once there is calculation overflow, the averge losses for large num step stays at the last updated value utill the algorithm stops. Thus the plots has relatively horizontal line segment on the graph.

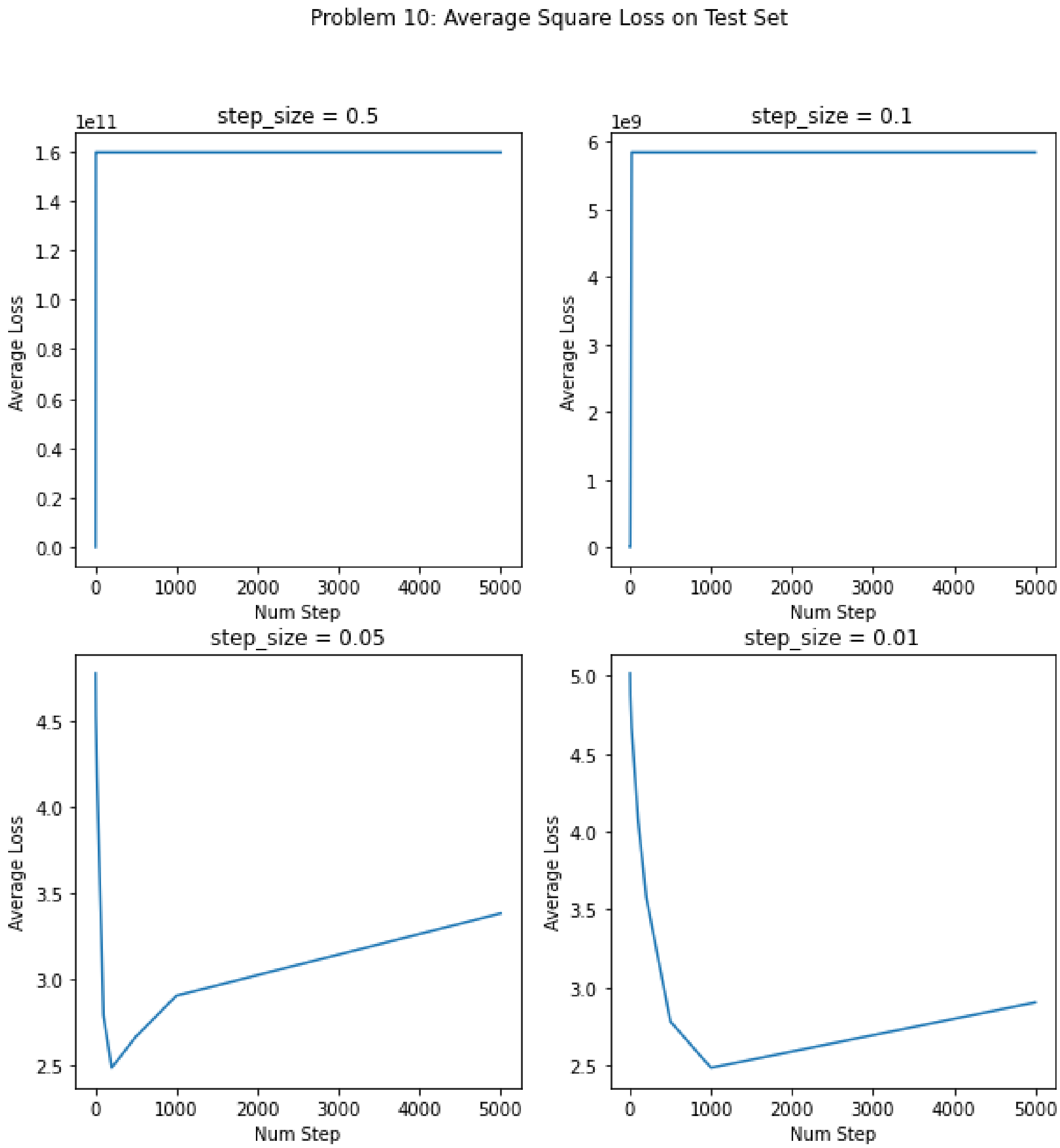
The plots for step sizes 0.5 and 0.1 quickly diverges and the error greatly increases as the number of steps used in the batch gradient descent function increases. Moreover, the larger step size 0.5 causes overflow in calculation quicker than 0.1. For smaller step sizes 0.05 and 0.01, the batch gradient descent converges, and the larger one 0.05 converges quicker than 0.01. In batch gradient descent, there may be a threshold between step sizes which lead to convergence or divergence. Larger step sizes may cause the function to get over the optimial point and thus lead to divergence.

Problem 10

[11]:
```
fig, ax = plt.subplots(2, 2, figsize=(10, 10))
steps = [0.5, 0.1, 0.05, 0.01]
numstep = [1, 5, 20, 100, 200, 500, 1000, 5000]
i = 0
for s in steps:
    testloss = []
    for n in numstep:
        theta_hist, loss_hist = batch_grad_descent(X_train, y_train, s, n, True)
        theta = [e for e in theta_hist if e[0]!= 0][-1]
        testloss.append(compute_square_loss(X_test, y_test, theta))
    ax[i//2, i%2].plot(numstep, testloss)
    ax[i//2, i%2].set_xlabel('Num Step')
    ax[i//2, i%2].set_ylabel('Average Loss')
    ax[i//2, i%2].set_title('step_size = {}'.format(s))
    i += 1
fig.suptitle('Problem 10: Average Square Loss on Test Set')
```

```
Overflow step size 0.5 and num step 5
Overflow step size 0.5 and num step 20
Overflow step size 0.5 and num step 100
Overflow step size 0.5 and num step 200
Overflow step size 0.5 and num step 500
Overflow step size 0.5 and num step 1000
Overflow step size 0.5 and num step 5000
Overflow step size 0.1 and num step 20
Overflow step size 0.1 and num step 100
Overflow step size 0.1 and num step 200
Overflow step size 0.1 and num step 500
Overflow step size 0.1 and num step 1000
Overflow step size 0.1 and num step 5000
```

[11]:
```
Text(0.5, 0.98, 'Problem 10: Average Square Loss on Test Set')
```



When step size = 0.5 or 0.1, the algorithm diveregs too quickly in both the training and the test with the tested number of steps. With smaller step size = 0.05 or 0.01, the error first decreases and then increases because of overfitting.

5 Ridge Regression

Problem 11

The gradient for the regularized objective function is

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

The expression for updating θ in the gradient descent algorithm for a step size η is

$$\theta^{t+1} = \theta^t - \eta^t \nabla J_\lambda(\theta) = \theta^t - \eta^t \frac{2}{m} X^T(X(\theta^T)^t - y) - \eta^t 2\lambda(\theta^T)^t$$

Problem 12

[12]:
```
def compute_regularized_square_loss(X, y, theta, lambda_reg):
    """
    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
    """
    m = len(X)
    loss = LA.norm(X @ theta.T - y.T) **2 / m + 2 * LA.norm(theta.T)
    return loss
```

[13]:
```
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)
    """
    m = len(X)
    grad = 2 * (X.T @ X @ theta.T - X.T @ y.T) / m + 2 * lambda_reg * theta.T
    return grad
```

Problem 13

[14]:
```
def regularized_grad_descent(X, y, alpha, lambda_reg, num_step):
    """
    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

    theta = np.zeros(num_features)  #Initialize theta
    theta_hist[0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta)

    grad_check = True

    for s in range(1, num_step+1):
        theta_hist[s] = theta_hist[s-1] - alpha *␣
↪compute_regularized_square_loss_gradient(X, y, theta_hist[s-1], lambda_reg)
        loss_hist[s] = compute_square_loss(X, y, theta_hist[s])
        if grad_check:
            if grad_checker(X, y, theta_hist[s]) == False:
                print('Overflow lambda', lambda_reg, 'and num step', num_step)
                break

    return theta_hist, loss_hist
```

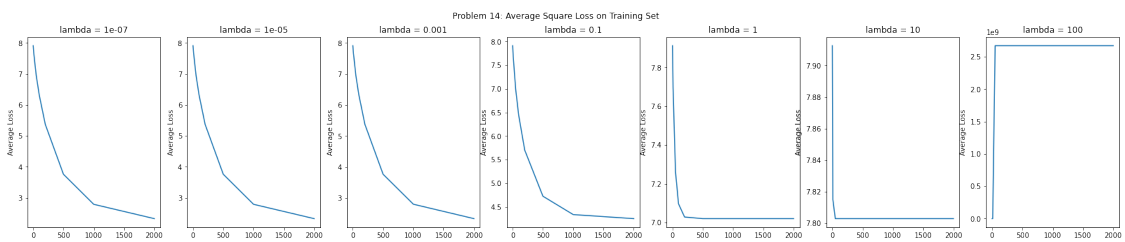
Problem 14

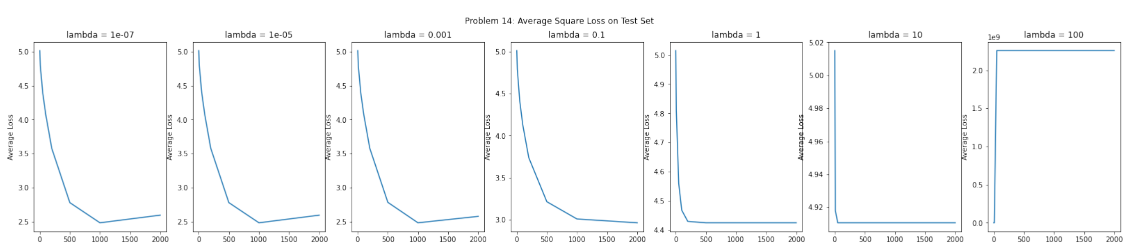
[15]:
```
fig1, ax1 = plt.subplots(1, 7, figsize=(28, 5))
fig2, ax2 = plt.subplots(1, 7, figsize=(28, 5))
step = 0.01
numstep = [1, 10, 50, 100, 200, 500, 1000, 2000]
lambda_reg = [10**(-7), 10**(-5), 10**(-3), 10**(-1), 1, 10, 100]
i = 0
for l in lambda_reg:
    loss = []
```

```
    testloss = []
    for n in numstep:
        theta_hist, loss_hist = regularized_grad_descent(X_train, y_train, step, l, n)
        temp = [e for e in loss_hist if e!= 0][-1]
        loss.append(temp)
        theta = [e for e in theta_hist if e[0]!= 0][-1]
        testloss.append(compute_square_loss(X_test, y_test, theta))
    ax1[i].plot(numstep, loss)
    ax1[i].set_xlabel('Num Step')
    ax1[i].set_ylabel('Average Loss')
    ax1[i].set_title('lambda = {}'.format(l))
    ax2[i].plot(numstep, testloss)
    ax2[i].set_xlabel('Num Step')
    ax2[i].set_ylabel('Average Loss')
    ax2[i].set_title('lambda = {}'.format(l))
    i += 1
fig1.suptitle('Problem 14: Average Square Loss on Training Set')
fig2.suptitle('Problem 14: Average Square Loss on Test Set')
```

```
Overflow lambda 100 and num step 50
Overflow lambda 100 and num step 100
Overflow lambda 100 and num step 200
Overflow lambda 100 and num step 500
Overflow lambda 100 and num step 1000
Overflow lambda 100 and num step 2000
```

[15]: Text(0.5, 0.98, 'Problem 14: Average Square Loss on Test Set')





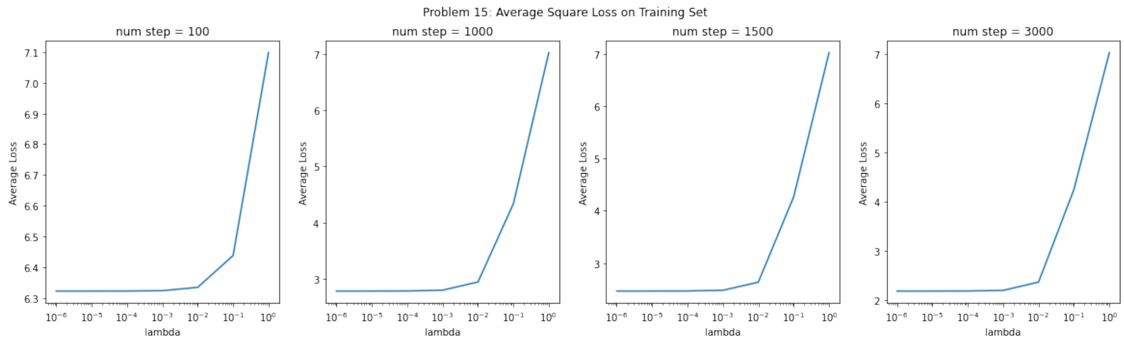
Note on the plot for lambda 100 on training set, because the grad_checker notifies the function to stop updating the theta and loss once there is calculation overflow, the averge losses for large num step stays at the last updated value utill the algorithm stops. It can be seen that the loss increases first and then the overflow error happens.

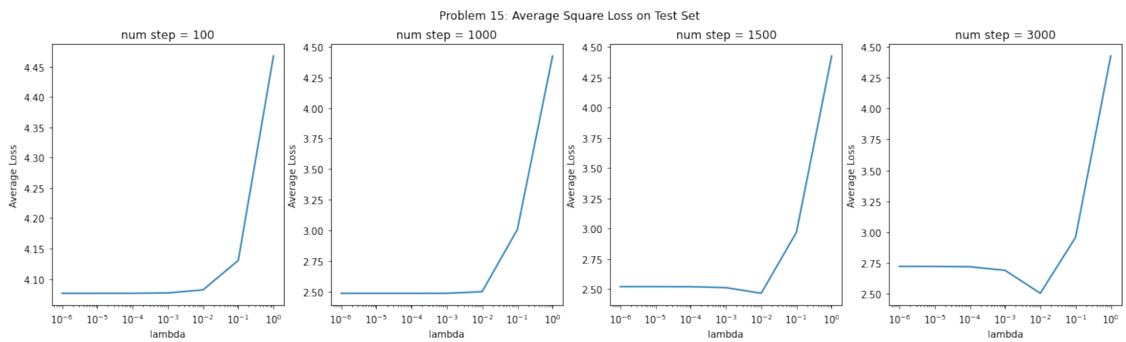
When lambda increases, the number of iterations at which overfitting starts increases, i.e. lambda helps reduce overfitting. However, when lambda is too large, the model has he risk of being underfitted, and thus the test error greatly increases.

Problem 15

[16]:
```
fig1, ax1 = plt.subplots(1, 4, figsize=(20, 5))
fig2, ax2 = plt.subplots(1, 4, figsize=(20, 5))
step = 0.01
numstep = [100, 1000, 1500, 3000]
lambda_reg = [10**(-6), 10**(-5), 10**(-4), 10**(-3), 10**(-2), 10**(-1), 1]
i = 0
for n in numstep:
    loss = []
    testloss = []
    theta_record = []
    for l in lambda_reg:
        theta_hist, loss_hist = regularized_grad_descent(X_train, y_train,␣
 ↪step, l, n)
        temp = [e for e in loss_hist if e!= 0][-1]
        loss.append(temp)
        theta = [e for e in theta_hist if e[0]!= 0][-1]
        testloss.append(compute_square_loss(X_test, y_test, theta))
    ax1[i].plot(lambda_reg, loss)
    ax1[i].set_xlabel('lambda')
    ax1[i].set_ylabel('Average Loss')
    ax1[i].set_title('num step = {}'.format(n))
    ax1[i].set_xscale('log')
    ax2[i].plot(lambda_reg, testloss)
    ax2[i].set_xlabel('lambda')
    ax2[i].set_ylabel('Average Loss')
    ax2[i].set_title('num step = {}'.format(n))
    ax2[i].set_xscale('log')
    i += 1
fig1.suptitle('Problem 15: Average Square Loss on Training Set')
fig2.suptitle('Problem 15: Average Square Loss on Test Set')
```

[16]:
```
Text(0.5, 0.98, 'Problem 15: Average Square Loss on Test Set')
```





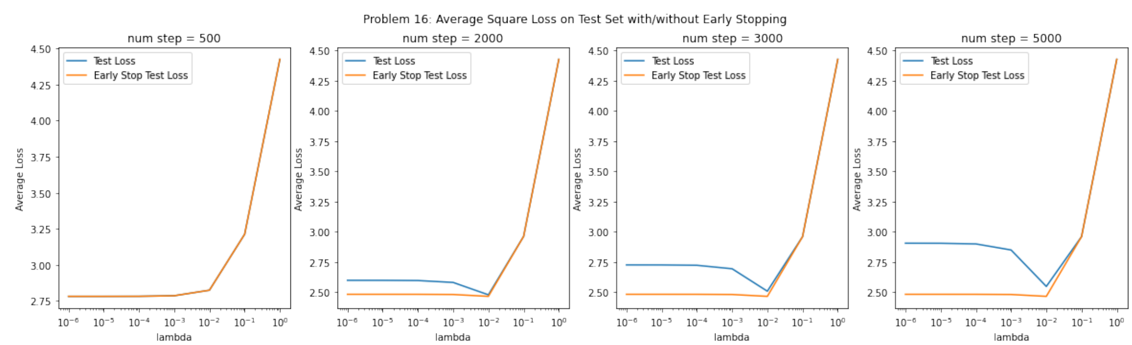
We may want to choose lambda = 0.01 under our settings and the tested values. When lambda = 0.01, the average loss on both the test and the training sets at multiple num steps are relatively small, and the overfitting were reduced as well. At larger number of steps the test error reaches a minimum at lambda = 0.01.

Problem 16

[17]:
```python
# I re-run the entire algorithm to separate my answers for each question.
fig2, ax2 = plt.subplots(1, 4, figsize=(20, 5))
step = 0.01
numstep = [500, 2000, 3000, 5000]
lambda_reg = [10**(-6), 10**(-5), 10**(-4), 10**(-3), 10**(-2), 10**(-1), 1]
i = 0
for n in numstep:
    loss = []
    testloss = []
    theta_temp = []
    best_loss = []
    for l in lambda_reg:
        theta_hist, loss_hist = regularized_grad_descent(X_train, y_train, step, l, n)
        theta = [e for e in theta_hist if e[0]!= 0][-1]
        theta_temp.append(theta)
```

```
        testloss.append(compute_square_loss(X_test, y_test, theta))
        best_loss.append(np.min([compute_square_loss(X_test, y_test, e) for e in theta_hist if e[0]!=0]))

    ax2[i].plot(lambda_reg, testloss, label = 'Test Loss')
    ax2[i].plot(lambda_reg, best_loss, label = 'Early Stop Test Loss')
    ax2[i].set_xlabel('lambda')
    ax2[i].set_ylabel('Average Loss')
    ax2[i].set_title('num step = {}'.format(n))
    ax2[i].set_xscale('log')
    ax2[i].legend(('Test Loss', 'Early Stop Test Loss'), loc='upper left')
    i += 1

fig2.suptitle('Problem 16: Average Square Loss on Test Set with/without Early Stopping')
```

[17]: Text(0.5, 0.98, 'Problem 16: Average Square Loss on Test Set with/without Early Stopping')



From the plot, we can still choose the same lambda as before, but the choice can be altered if more lambda values are tested.

Problem 17

In practice we want to select the lambda that gives the lowest testing error with acceptable or relatively small training error, because we want to generalize the model. Sometimes low training error may shows overfitting, and thus gives larger test errors. After exploring the optimal lambda value, we can run through different number of steps and step sizes that also leads to minimal test error as possible. Then with these hyperparameters, we can decide the corresponding theta to use.

6 Stochastic Gradient Descent (Optional)

Problem 18

The objective function

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

can be written as

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

since SGD picks one point at a time and $\frac{1}{m}(m(\lambda\theta^T\theta)) = \lambda\theta^T\theta$

Thus,

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

with objective function

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

Problem 19

With our objective function written inside the summation, the gradient can be written as

$$\nabla J_\lambda(\theta) = \nabla\frac{1}{m}\sum_{i=1}^{m} f_i(\theta)$$

Since gradient is a linear operation,

$$\nabla J_\lambda(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(\theta)$$

Recall the expected value formula, the expected value of this gradient is the sum of all possible choices of i, which is sampled uniformly at random from {1, ..., m}

$$E[\nabla f_i(\theta)] = \nabla\sum_{j=1}^{m} P(j = i) f_i(\theta)$$

We can also move the gradient into the summation,

$$E[\nabla f_i(\theta)] = \frac{1}{m}\sum_{j=1}^{m} f_i(\theta)$$

Therefore,

$$E[\nabla f_i(\theta)] = \nabla J_\lambda(\theta)$$

Problem 20

From the previous problems, we can write the gradient of our $f_i(\theta)$ as

$$\begin{aligned}\nabla f_i(\theta) &= \nabla((h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta) \\ &= 2(h_\theta(x_i) - y_i)\nabla h_\theta(x_i) + 2\lambda\theta \\ &= 2((h_\theta(x_i) - y_i)x_i + \lambda\theta)\end{aligned}$$

The expression for updating θ in the gradient descent algorithm for a step size η is

$$\theta^{t+1} = \theta^t - \eta^t \nabla J_\lambda(\theta) = \theta^t - \eta^t \nabla E[\nabla f_i(\theta)]$$

then we can plug in our formula for $\nabla f_i(\theta)$ and its expected value.

Problem 21

[ ]:
```
def compute_stochastic_gradient(X, y, theta, lambda_reg):
    grad = 2 * ((theta.T @ X) - y) * X + 2 * lambda_reg * theta
    return grad
```

[32]:
```
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2,␣
↪num_epoch=1000, eta0=False):
    """
    In this question you will implement stochastic gradient descent with␣
↪regularization term

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

    Returns:
        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

    theta_hist[0][0] = theta
    loss_hist[0] = compute_square_loss(X, y, theta)
```

```
    for i in range(1, num_epoch):
        randomidx = np.random.permutation(num_instances)

        for j in randomidx:
            Xi = X[j]
            yi = y[j]
            theta = theta - alpha * compute_stochastic_gradient(Xi, yi, theta, lambda_reg)
            theta_hist[i][j] = theta
            loss_hist[i][j] = compute_square_loss(X, y, theta) + lambda_reg * (theta.T @ theta)

    return theta_hist, loss_hist
```

2 Image Classification with Regularized Logistic Regression

7 Logistic Regression

Problem 23

[18]:
```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

[19]:
```
def pre_process_mnist_01():
    """
    Load the mnist datasets, selects the classes 0 and 1
    and normalize the data.
    Args: none
    Outputs:
        X_train: np.array of size (n_training_samples, n_features)
        X_test: np.array of size (n_test_samples, n_features)
        y_train: np.array of size (n_training_samples)
        y_test: np.array of size (n_test_samples)
    """
    X_mnist, y_mnist = fetch_openml('mnist_784', version=1,
                                    return_X_y=True, as_frame=False)
    indicator_01 = (y_mnist == '0') + (y_mnist == '1')
    X_mnist_01 = X_mnist[indicator_01]
    y_mnist_01 = y_mnist[indicator_01]
    X_train, X_test, y_train, y_test = train_test_split(X_mnist_01, y_mnist_01,
                                                        test_size=0.33,
                                                        shuffle=False)
```

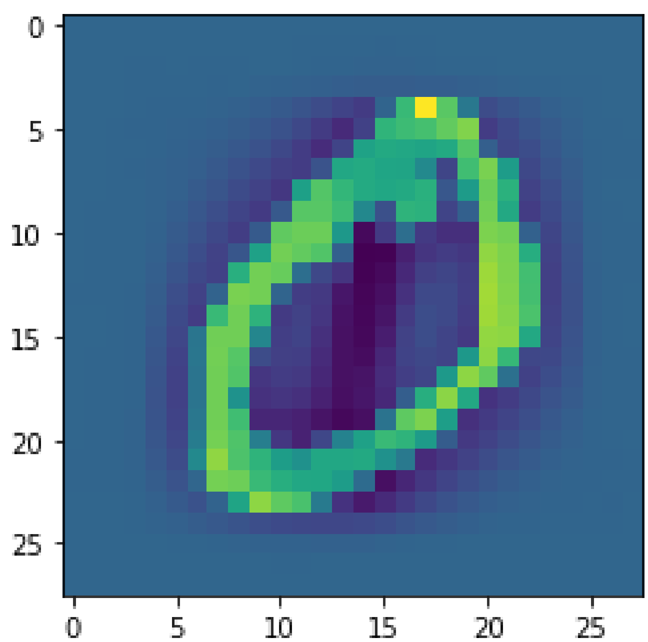
```
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    y_test = 2 * np.array([int(y) for y in y_test]) - 1
    y_train = 2 * np.array([int(y) for y in y_train]) - 1
    return X_train, X_test, y_train, y_test
```

[20]:
```
def sub_sample(N_train, X_train, y_train):
    """
    Subsample the training data to keep only N first elements
    Args: none
    Outputs:
        X_train: np.array of size (n_training_samples, n_features)
        X_test: np.array of size (n_test_samples, n_features)
        y_train: np.array of size (n_training_samples)
        y_test: np.array of size (n_test_samples)
    """
    assert N_train <= X_train.shape[0]
    return X_train[:N_train, :], y_train[:N_train]
```

[21]:
```
X_train, X_test, y_train, y_test = pre_process_mnist_01()
demoX, demoy = sub_sample(1, X_train, y_train)
```

[22]:
```
plt.imshow(demoX.reshape((28, 28)))
```

[22]: <matplotlib.image.AxesImage at 0x227476248b0>



The general formula of log-likelihood on each data point can be written as a function of the margin,

$$l(\theta) = log(1 + e^{-margin})$$

where margin is defined as $yh_{\theta,b}(x)$.

Then the average on m observations is

$$L(\theta) = \frac{1}{m}\sum_{i=1}^{m} log(1 + e^{-margin})$$

This problem is a binary classification with $y_i \in \{-1, 1\}$, and thus the logistic loss can be written in log-likelihood as

$$\begin{aligned}
L(\theta) &= \frac{1}{m}\sum_{i=1}^{m}(\frac{1}{2}(1+y_i)log(1+e^{-margin_{y=1}}) + \frac{1}{2}(1-y_i)log(1+e^{-margin_{y=-1}})) \\
&= \frac{1}{2m}\sum_{i=1}^{m}((1+y_i)log(1+e^{-margin_{y=1}}) + (1-y_i)log(1+e^{-margin_{y=-1}})) \\
&= \frac{1}{2m}\sum_{i=1}^{m}((1+y_i)log(1+e^{-(1)h_{\theta,b}(x_i)}) + (1-y_i)log(1+e^{-(-1)h_{\theta,b}(x_i)})) \\
&= \frac{1}{2m}\sum_{i=1}^{m}((1+y_i)log(1+e^{-h_{\theta,b}(x_i)}) + (1-y_i)log(1+e^{h_{\theta,b}(x_i)}))
\end{aligned}$$

Problem 24

The loss function if we regularize the coefficients with l_1 penalty will be

$$L_\alpha(\theta) = L(\theta) + \alpha|\theta| = \frac{1}{2m}\sum_{i=1}^{m}((1+y_i)log(1+e^{-h_{\theta,b}(x_i)}) + (1-y_i)log(1+e^{h_{\theta,b}(x_i)})) + \alpha|\theta|$$

where $|\theta|$ is the l_1 norm.

Problem 25

[23]:
```
def classification_error(clf, X, y):
    y_pred = clf.predict(X)
    err = np.sum([i != j for i, j in zip(y_pred, y)])/len(y)
    return err
```

[25]:
```
clf = SGDClassifier(loss='log', max_iter=1000,
                    tol=1e-3,
                    penalty='l1', alpha=0.01,
                    learning_rate='invscaling',
                    power_t=0.5,
                    eta0=0.01,
                    verbose=1)
```

```
clf.fit(X_train, y_train)
ce = classification_error(clf, X_test, y_test)
check = 1 - clf.score(X_test, y_test)
print('The calculated error is equal to the check:', abs(ce-check)<0.001)
```

```
-- Epoch 1
Norm: 0.69, NNZs: 280, Bias: -0.007444, T: 9902, Avg. loss: 0.041943
Total training time: 0.04 seconds.
-- Epoch 2
Norm: 0.78, NNZs: 257, Bias: -0.007568, T: 19804, Avg. loss: 0.031704
Total training time: 0.07 seconds.
-- Epoch 3
Norm: 0.84, NNZs: 242, Bias: -0.007389, T: 29706, Avg. loss: 0.029940
Total training time: 0.11 seconds.
-- Epoch 4
Norm: 0.89, NNZs: 233, Bias: -0.007072, T: 39608, Avg. loss: 0.029190
Total training time: 0.14 seconds.
-- Epoch 5
Norm: 0.92, NNZs: 225, Bias: -0.006699, T: 49510, Avg. loss: 0.028613
Total training time: 0.17 seconds.
-- Epoch 6
Norm: 0.96, NNZs: 220, Bias: -0.006284, T: 59412, Avg. loss: 0.028295
Total training time: 0.21 seconds.
-- Epoch 7
Norm: 0.99, NNZs: 212, Bias: -0.005864, T: 69314, Avg. loss: 0.027897
Total training time: 0.25 seconds.
-- Epoch 8
Norm: 1.01, NNZs: 205, Bias: -0.005431, T: 79216, Avg. loss: 0.027647
Total training time: 0.29 seconds.
Convergence after 8 epochs took 0.29 seconds
The calculated error is equal to the check: True
```

Problem 26

[26]:
```
alp = np.linspace(10**(-4), 10**(-1), num=10)
avg = []
std = []
for value in alp:
    clf = SGDClassifier(loss='log', max_iter=1000,
                        tol=1e-3,
                        penalty='l1', alpha=value,
                        learning_rate='invscaling',
                        power_t=0.5,
                        eta0=0.01,
                        verbose=1)
    resX_train, resy_train = sub_sample(100, X_train, y_train)

    record = []
```

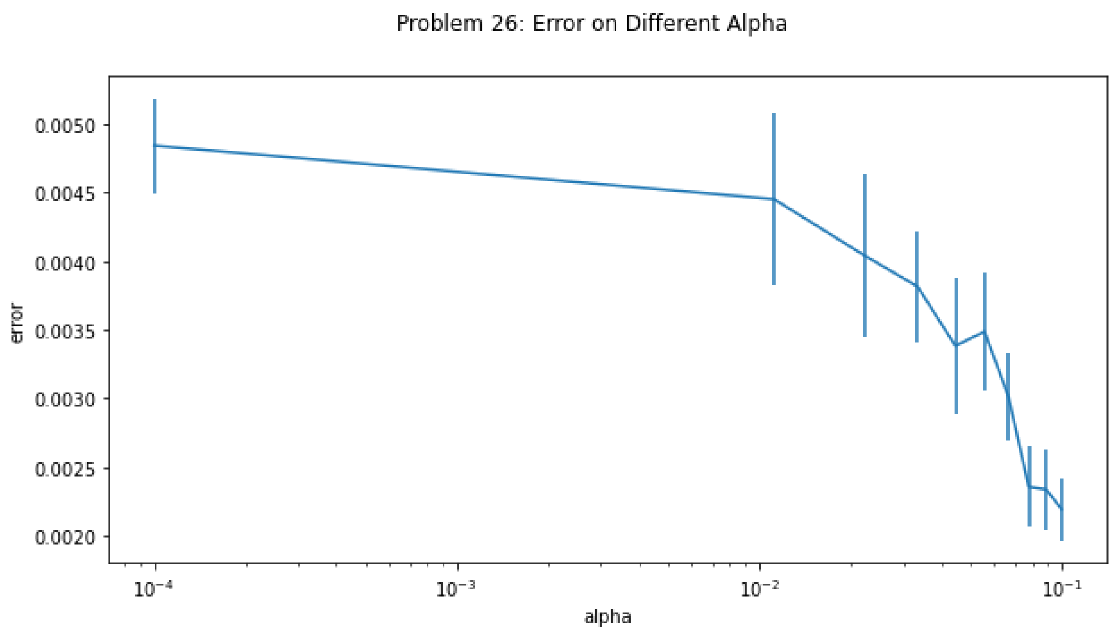
```python
    for i in range(10):
        clf.fit(resX_train, resy_train)
        ce = classification_error(clf, X_test, y_test)
        record.append(ce)
    avg.append(np.mean(record))
    std.append(np.std(record))
```

```
-- Epoch 1
Norm: 0.30, NNZs: 606, Bias: 0.015959, T: 100, Avg. loss: 0.159348
Total training time: 0.00 seconds.
-- Epoch 2
Norm: 0.34, NNZs: 606, Bias: 0.017337, T: 200, Avg. loss: 0.070290
Total training time: 0.00 seconds.
-- Epoch 3
Norm: 0.36, NNZs: 606, Bias: 0.018078, T: 300, Avg. loss: 0.056267
Total training time: 0.00 seconds.
-- Epoch 4
Norm: 0.38, NNZs: 607, Bias: 0.018640, T: 400, Avg. loss: 0.048849
Total training time: 0.00 seconds.
-- Epoch 5
Norm: 0.39, NNZs: 606, Bias: 0.019063, T: 500, Avg. loss: 0.044003
Total training time: 0.00 seconds.
-- Epoch 6
Norm: 0.41, NNZs: 607, Bias: 0.019406, T: 600, Avg. loss: 0.040490
Total training time: 0.00 seconds.
-- Epoch 7
Norm: 0.41, NNZs: 605, Bias: 0.019693, T: 700, Avg. loss: 0.037780
Total training time: 0.00 seconds.
-- Epoch 8
Norm: 0.42, NNZs: 606, Bias: 0.019943, T: 800, Avg. loss: 0.035606
Total training time: 0.00 seconds.
-- Epoch 9
Norm: 0.43, NNZs: 607, Bias: 0.020162, T: 900, Avg. loss: 0.033810
Total training time: 0.00 seconds.
-- Epoch 10
Norm: 0.44, NNZs: 607, Bias: 0.020351, T: 1000, Avg. loss: 0.032283
Total training time: 0.01 seconds.
-- Epoch 11
Norm: 0.44, NNZs: 606, Bias: 0.020531, T: 1100, Avg. loss: 0.030969
Total training time: 0.01 seconds.
-- Epoch 12
Norm: 0.45, NNZs: 606, Bias: 0.020697, T: 1200, Avg. loss: 0.029818
Total training time: 0.01 seconds.
-- Epoch 13
Norm: 0.45, NNZs: 607, Bias: 0.020845, T: 1300, Avg. loss: 0.028802
Total training time: 0.01 seconds.
-- Epoch 14
Norm: 0.46, NNZs: 606, Bias: 0.020981, T: 1400, Avg. loss: 0.027891
```

```
Total training time: 0.01 seconds.
-- Epoch 12
Norm: 0.60, NNZs: 127, Bias: 0.037758, T: 1200, Avg. loss: 0.181046
Total training time: 0.01 seconds.
-- Epoch 13
Norm: 0.61, NNZs: 127, Bias: 0.038725, T: 1300, Avg. loss: 0.179960
Total training time: 0.01 seconds.
-- Epoch 14
Norm: 0.62, NNZs: 124, Bias: 0.039652, T: 1400, Avg. loss: 0.179518
Total training time: 0.01 seconds.
-- Epoch 15
Norm: 0.63, NNZs: 124, Bias: 0.040547, T: 1500, Avg. loss: 0.178836
Total training time: 0.01 seconds.
-- Epoch 16
Norm: 0.64, NNZs: 121, Bias: 0.041410, T: 1600, Avg. loss: 0.178159
Total training time: 0.01 seconds.
-- Epoch 17
Norm: 0.65, NNZs: 117, Bias: 0.042243, T: 1700, Avg. loss: 0.177551
Total training time: 0.01 seconds.
-- Epoch 18
Norm: 0.66, NNZs: 116, Bias: 0.043050, T: 1800, Avg. loss: 0.176656
Total training time: 0.01 seconds.
Convergence after 18 epochs took 0.01 seconds
```

[28]:
```
fig, ax = plt.subplots(1, figsize=(10, 5))
ax.errorbar(alp, avg, std)
ax.set_xscale('log')
ax.set_xlabel('alpha')
ax.set_ylabel('error')
fig.suptitle('Problem 26: Error on Different Alpha')
```

[28]:
```
Text(0.5, 0.98, 'Problem 26: Error on Different Alpha')
```



Problem 27

The source of randomness we averaging over comes from the stochastic gradient descent algorithm, in which we randomly choose a data point to determine our step direction in each training iteration. Such way of choice may lead to different end coefficients to the model and thus gives different classification results and errors.

Problem 28

From the values we tested in this problem, the optimal value is

[90]:
```
print('The optimal alpha that gives the lowest classification error on average␣
↪is', alp[avg.index(np.min(avg))])
```

```
The optimal alpha that gives the lowest classification error on average is
0.0889
```

Problem 29

[29]:
```
# I fit the entire process again to make the separation of each problem clearer.
alp = np.linspace(10**(-4), 10**(-1), num=10)
theta = []
for value in alp:
    clf = SGDClassifier(loss='log', max_iter=1000,
                        tol=1e-3,
                        penalty='l1', alpha=value,
                        learning_rate='invscaling',
                        power_t=0.5,
                        eta0=0.01,
```

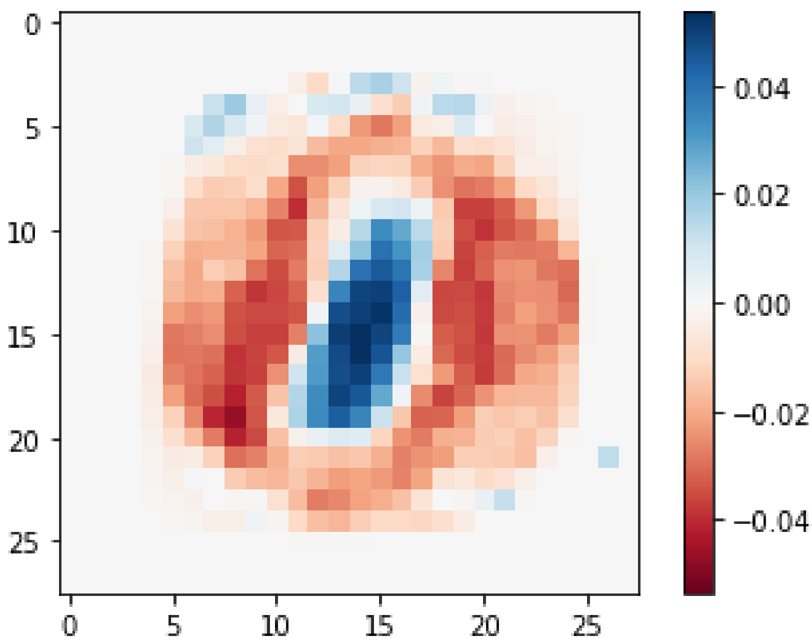
```
                    verbose=1)
    clf.fit(resX_train, resy_train)
    theta.append(clf.coef_)
```

```
-- Epoch 1
Norm: 0.31, NNZs: 607, Bias: 0.004007, T: 100, Avg. loss: 0.158902
Total training time: 0.00 seconds.
-- Epoch 2
Norm: 0.34, NNZs: 607, Bias: 0.005472, T: 200, Avg. loss: 0.072292
Total training time: 0.00 seconds.
-- Epoch 3
Norm: 0.37, NNZs: 607, Bias: 0.006293, T: 300, Avg. loss: 0.057935
Total training time: 0.00 seconds.
-- Epoch 4
Norm: 0.38, NNZs: 607, Bias: 0.006885, T: 400, Avg. loss: 0.050280
Total training time: 0.00 seconds.
-- Epoch 5
Norm: 0.40, NNZs: 607, Bias: 0.007347, T: 500, Avg. loss: 0.045289
Total training time: 0.00 seconds.
-- Epoch 6
Norm: 0.41, NNZs: 607, Bias: 0.007730, T: 600, Avg. loss: 0.041684
Total training time: 0.00 seconds.
-- Epoch 7
Norm: 0.42, NNZs: 607, Bias: 0.008044, T: 700, Avg. loss: 0.038890
Total training time: 0.00 seconds.
-- Epoch 8
Norm: 0.42, NNZs: 606, Bias: 0.008323, T: 800, Avg. loss: 0.036651
Total training time: 0.00 seconds.
-- Epoch 9
Norm: 0.43, NNZs: 606, Bias: 0.008564, T: 900, Avg. loss: 0.034794
Total training time: 0.00 seconds.
-- Epoch 10
Norm: 0.44, NNZs: 606, Bias: 0.008781, T: 1000, Avg. loss: 0.033219
Total training time: 0.00 seconds.
-- Epoch 11
Norm: 0.44, NNZs: 607, Bias: 0.008980, T: 1100, Avg. loss: 0.031864
Total training time: 0.00 seconds.
-- Epoch 12
Norm: 0.45, NNZs: 607, Bias: 0.009162, T: 1200, Avg. loss: 0.030672
Total training time: 0.00 seconds.
-- Epoch 13
Norm: 0.45, NNZs: 606, Bias: 0.009329, T: 1300, Avg. loss: 0.029619
Total training time: 0.01 seconds.
-- Epoch 14
Norm: 0.46, NNZs: 605, Bias: 0.009484, T: 1400, Avg. loss: 0.028677
Total training time: 0.01 seconds.
-- Epoch 15
Norm: 0.46, NNZs: 605, Bias: 0.009626, T: 1500, Avg. loss: 0.027826
```

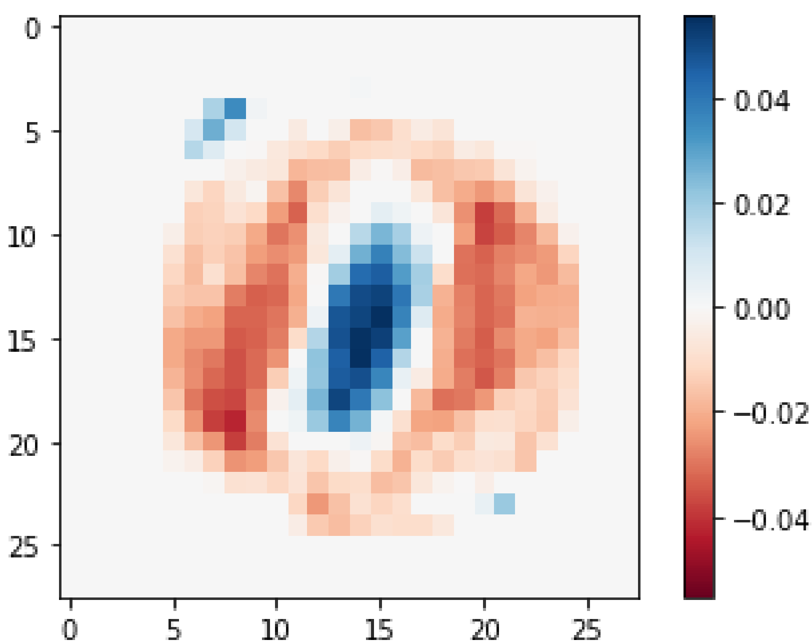
```
Total training time: 0.00 seconds.
-- Epoch 7
Norm: 0.53, NNZs: 145, Bias: 0.019457, T: 700, Avg. loss: 0.188052
Total training time: 0.00 seconds.
-- Epoch 8
Norm: 0.55, NNZs: 142, Bias: 0.020809, T: 800, Avg. loss: 0.186412
Total training time: 0.00 seconds.
-- Epoch 9
Norm: 0.56, NNZs: 137, Bias: 0.022065, T: 900, Avg. loss: 0.184973
Total training time: 0.00 seconds.
-- Epoch 10
Norm: 0.58, NNZs: 134, Bias: 0.023223, T: 1000, Avg. loss: 0.183904
Total training time: 0.00 seconds.
-- Epoch 11
Norm: 0.59, NNZs: 134, Bias: 0.024327, T: 1100, Avg. loss: 0.182683
Total training time: 0.01 seconds.
-- Epoch 12
Norm: 0.60, NNZs: 132, Bias: 0.025375, T: 1200, Avg. loss: 0.181683
Total training time: 0.01 seconds.
-- Epoch 13
Norm: 0.62, NNZs: 129, Bias: 0.026373, T: 1300, Avg. loss: 0.181279
Total training time: 0.01 seconds.
-- Epoch 14
Norm: 0.63, NNZs: 126, Bias: 0.027336, T: 1400, Avg. loss: 0.180459
Total training time: 0.01 seconds.
-- Epoch 15
Norm: 0.64, NNZs: 124, Bias: 0.028259, T: 1500, Avg. loss: 0.179808
Total training time: 0.01 seconds.
-- Epoch 16
Norm: 0.65, NNZs: 123, Bias: 0.029155, T: 1600, Avg. loss: 0.179273
Total training time: 0.01 seconds.
Convergence after 16 epochs took 0.01 seconds
```

[31]:
```python
for i in range(len(theta)):
    print('Alpha = ', alp[i])
    scale = np.abs(theta[i]).max()
    plt.imshow(theta[i].reshape((28,28)),cmap=plt.cm.RdBu, vmax=scale,␣
 ↪vmin=-scale)
    plt.colorbar()
    plt.show()
```

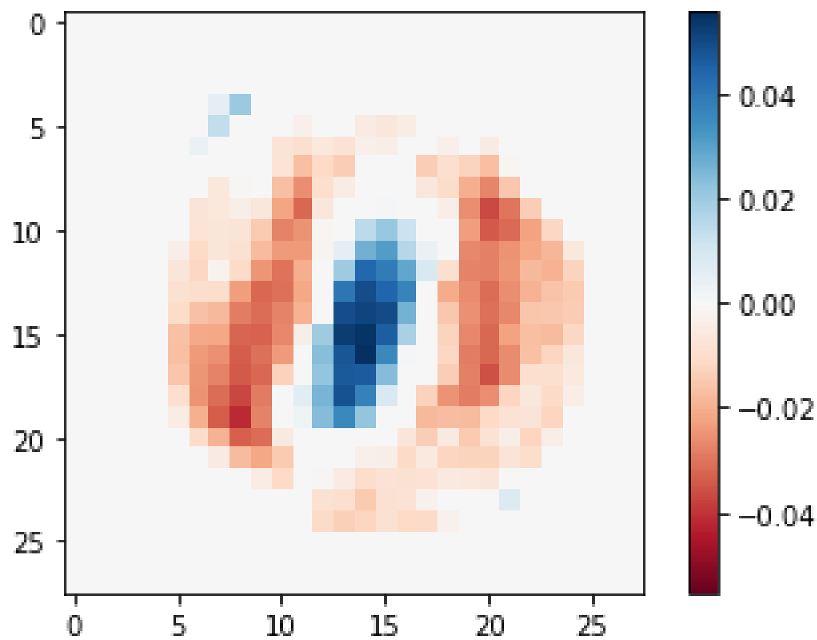
```
Alpha =  0.0001
```



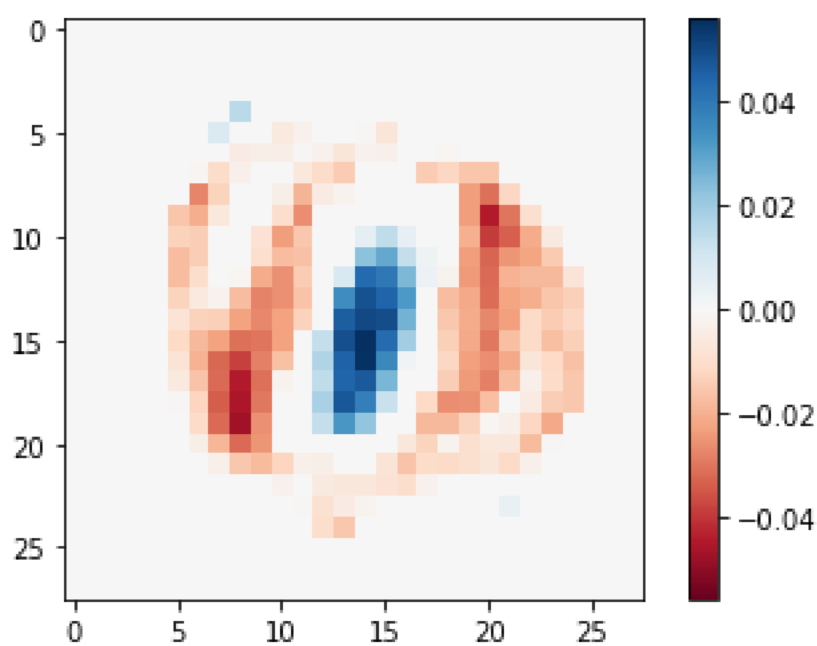
Alpha = 0.0112



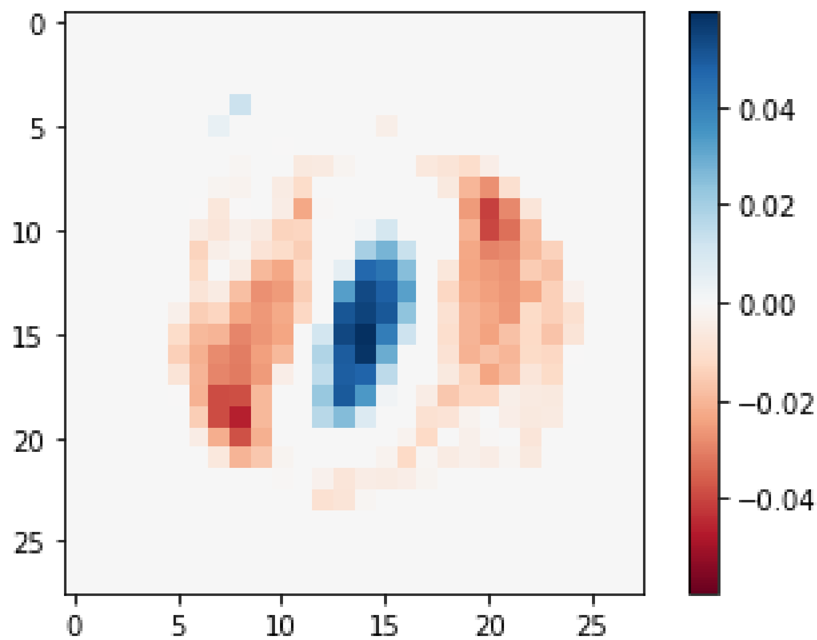
Alpha = 0.0223



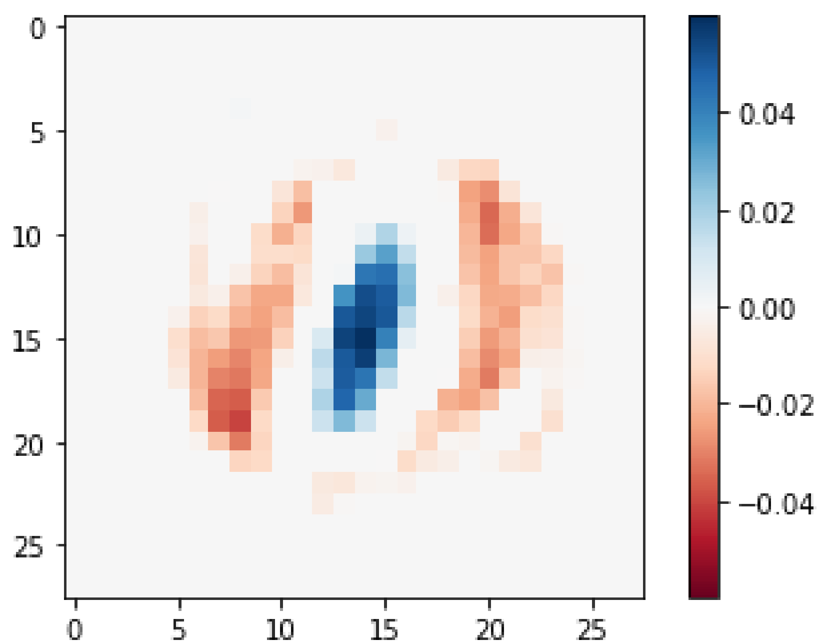
Alpha = 0.033400000000000006



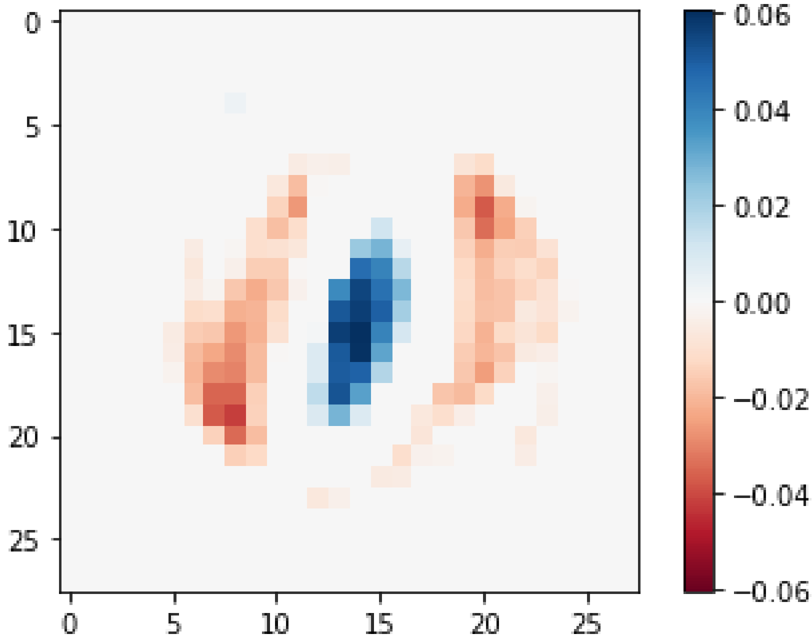
Alpha = 0.044500000000000005



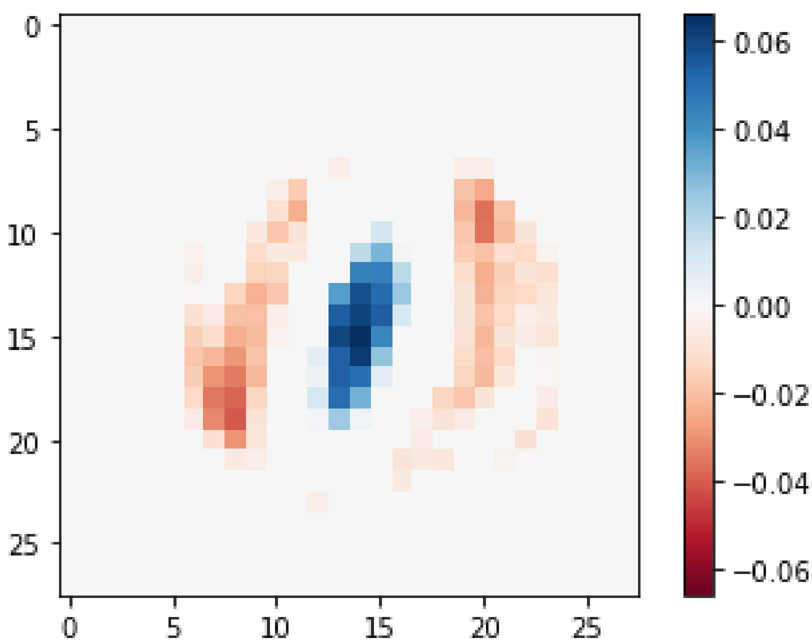
Alpha = 0.055600000000000004



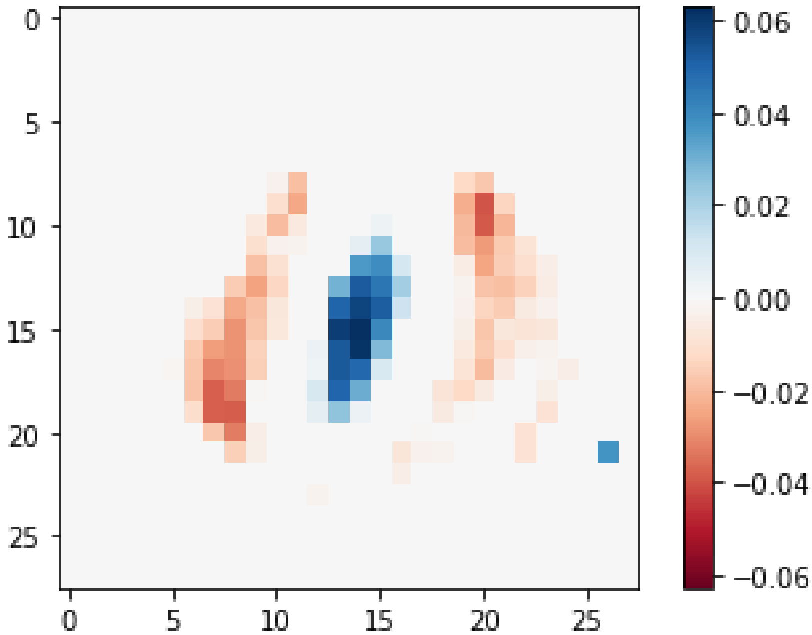
Alpha = 0.06670000000000001



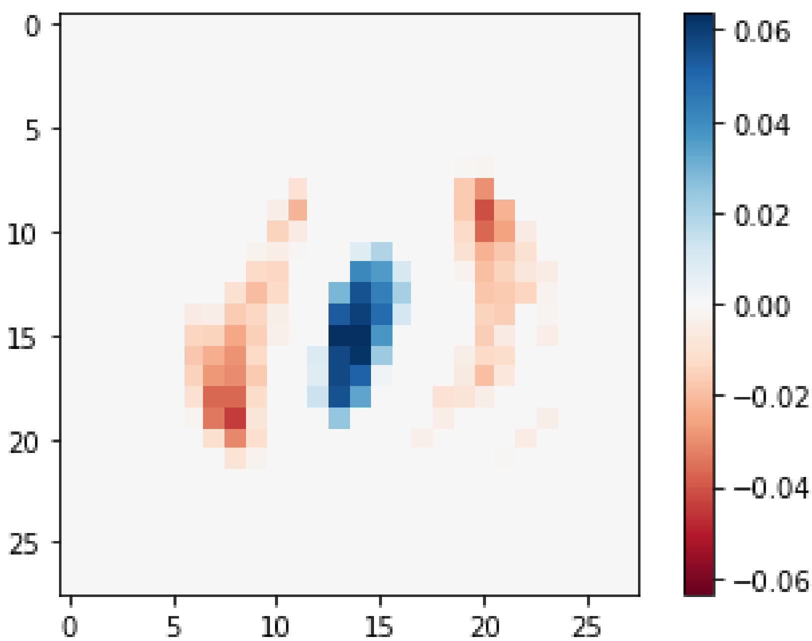

```
Alpha =  0.07780000000000001
```




```
Alpha =  0.0889
```



`Alpha =  0.1`



Problem 30

As alpha increases, on the images in the previous problem, areas with large magnitude, i.e. deep blue and deep red, decreases. This shows that less “extreme” values appear in the coefficients, and their values are more “centralized” because of larger penalty added by the alpha value or the regularization strength.