DSGA 1003 HW2 yp2201@nyu.edu

1. As defined, training error $= \frac{1}{N}\|X\hat{b} - y\|_2^2$

Also, $y_i = b^T x_i + \varepsilon_i$, $\hat{b} = (X^TX)^{-1}X^Ty$

$\Downarrow$

$y = Xb + \varepsilon$

$\therefore$ Training error $= \frac{1}{N}\|X\hat{b} - y\|_2^2 = \frac{1}{N}\|X(X^TX)^{-1}X^T(Xb+\varepsilon) - (Xb+\varepsilon)\|_2^2$

$= \frac{1}{N}\|X(X^TX)^{-1}X^TXb + X(X^TX)^{-1}X^T\varepsilon - Xb - \varepsilon\|_2^2$

$= \frac{1}{N}\|XX^{-1}X^{-T}X^TXb + X(X^TX)^{-1}X^T\varepsilon - Xb - \varepsilon\|_2^2$

$= I$ (cancel out)

$$\boxed{= \frac{1}{N}\|(X(X^TX)^{-1}X^T - I)\varepsilon\|_2^2}$$

2. - $A = X(X^TX)^{-1}X^T$,

- $A^TA = (X(X^TX)^{-1}X^T)^T (X(X^TX)^{-1}X^T) = ((X^TX)^{-1}X^T)^T X^T X (X^TX)^{-1}X^T$

$= X(X^TX)^{-T}X^TX(X^TX)^{-1}X^T = X((X^TX)^T)^{-1}X^TX(X^TX)^{-1}X^T$

$= X(X^TX)^{-1}X^TX(X^TX)^{-1}X^T = X\underbrace{X^{-1}}_{=I}\underbrace{X^{-T}}_{=I}X^TX(X^TX)^{-1}X^T = X(X^TX)^{-1}X^T = A$

- $A^2 = X(X^TX)^{-1}X^TX(X^TX)^{-1}X^T = X\underbrace{X^{-1}}_{=I}\underbrace{X^{-T}}_{=I}X^TX(X^TX)^{-1}X^T = X(X^TX)^{-1}X^T = A$

* - Expectation of the traing error:

$E\left[\frac{1}{N}\|(X(X^TX)^{-1}X^T - I)\epsilon\|_2^2\right] = E\left[\frac{1}{N}\|(A-I)\epsilon\|_2^2\right]$

$= E\left[\frac{1}{N}(A\epsilon - I\epsilon)^T(A\epsilon - I\epsilon)\right] = E\left[\frac{1}{N}(\epsilon^TA^T - \epsilon^TI^T)(A\epsilon - I\epsilon)\right]$

$= E\left[\frac{1}{N}(\epsilon^TA^TA\epsilon - \epsilon^TA^TI\epsilon - \epsilon^TI^TA\epsilon + \epsilon^TI^TI\epsilon)\right]$

(Since $A^T=A$, $A^2=A$, $A^TA=A$, $I^T=I$)

$= E\left[\frac{1}{N}(\epsilon^TA\epsilon - \epsilon^TA\epsilon - \epsilon^TA\epsilon + \epsilon^T\epsilon)\right] = E\left[\frac{1}{N}(-\epsilon^TA\epsilon + \epsilon^T\epsilon)\right]$

$= \frac{1}{N}(-\underbrace{E(\epsilon^TA\epsilon)}_{=d\sigma^2} + \underbrace{E(\epsilon^T\epsilon)}_{=N\sigma^2}) = \frac{1}{N}(-d\sigma^2 + N\sigma^2) = \boxed{\frac{(N-d)}{N}\sigma^2}$

---

* $E(\epsilon^T\epsilon) = E(\epsilon^2) = Var(\epsilon) - [E(\epsilon)]^2 = N\sigma^2 - 0 = N\sigma^2$.

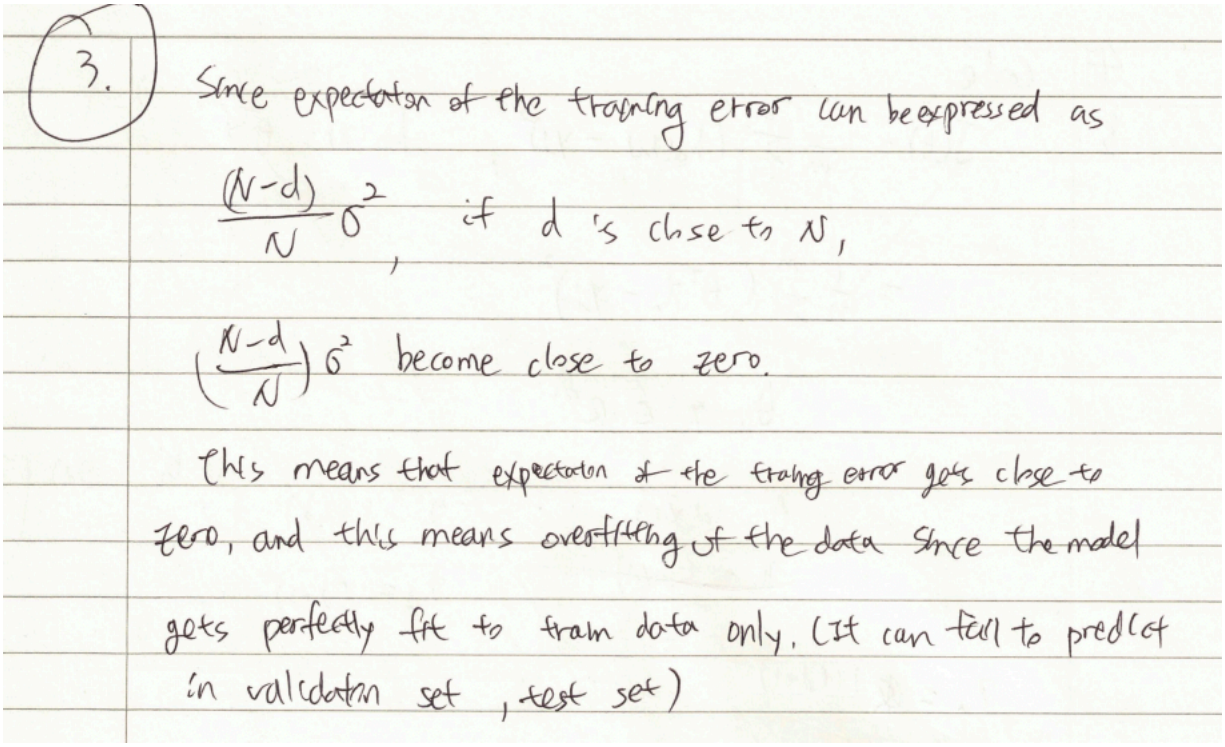
$E(\epsilon^TA\epsilon) = E\left[(\epsilon_1, \epsilon_2, \cdots, \epsilon_N)\begin{pmatrix}\epsilon_1\lambda_1\\ \epsilon_2\lambda_2\\ \vdots\\ \epsilon_N\lambda_N\end{pmatrix}\right] = E\left[(\epsilon_1^2\lambda_1 + \epsilon_2^2\lambda_2 + \cdots + \epsilon_N^2\lambda_N)\right] = E\left(\sum_{i=1}^{n}\epsilon_i^2\lambda_i\right)$

$= \sum_{i=1}^{d} E(\epsilon_i^2) = d\sigma^2$

Let $A = PDP^T$, where $D = \begin{bmatrix}\lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n\end{bmatrix}$,

$A^2 = PD^2P^T = A = PDP^T \Rightarrow D^2 = D$, $\underline{\lambda_i^2 = \lambda_i \ \forall i}$, $(\lambda_i - 1)\lambda_i = 0 \Rightarrow \lambda_i = 1$ or 0

Since A has rank d, $\begin{cases}\lambda_i = 1 : d\\ \lambda_i = 0 : (n-d)\end{cases}$



Q4.

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.

```
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

#ignore warnings
import warnings
warnings.filterwarnings('ignore')

%matplotlib inline

#####################################
### Feature normalization
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
    # discard features that are constant in the training set
    train = train[:, ~np.all(train[1:] == train[:-1], axis=0)]

    # normalizing
    train_normalized = (train - np.min(train, axis=0)) / (np.max(train, axis=0) - np.min(train, axis=0))
    test_normalized = (test - np.min(test, axis=0)) / (np.max(test, axis=0) - np.min(test, axis=0))

    return train_normalized, test_normalized
```

5. As $h_\theta(x) = \theta^T x$, $h_\theta(x_i) = x_i \cdot \theta$

$$\left(x_i \cdot \theta = [1 \; x_i \; x_i^2 \cdots x_i^d] \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \right)$$

∴, Objective function $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$.

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

$$= \frac{1}{m} \left((x_1\theta - y_1)^2 + (x_2\theta - y_2)^2 + \cdots + (x_m\theta - y_m)^2 \right)$$

$$= \frac{1}{m} \left(\sqrt{(x_1\theta - y_1)^2 + (x_2\theta - y_2)^2 + \cdots + (x_m\theta - y_m)^2} \right)^2$$

$$= \frac{1}{m} \left(\| X\theta - y \|_2 \right)^2 = \boxed{\frac{1}{m} \| X\theta - y \|_2^2}$$

6. Gradient of J:

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

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

$$= \frac{1}{m} \nabla_\theta \left(\theta^T X^T X \theta - \theta^T X^T y - y^T X \theta + y^T y \right)$$

$$= \frac{1}{m} \nabla_\theta \left(\theta^T X^T X \theta - 2\theta^T X^T y + y^T y \right)$$

$$= \frac{1}{m} \left(2X^T X\theta - 2X^T y \right) = \frac{2}{m} \left(X^T X\theta - X^T y \right)$$

7. Updating θ in the gradient descent algorithm, Step size = η

- Initialize: θ_0
- Repeat: $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$, where $\nabla_\theta J(\theta) = \frac{2}{m}(X^T X\theta - X^T y)$

$$\left(\Rightarrow \theta_{i+1} = \theta_i - \eta \cdot \left(\frac{2}{m} (X^T X\theta_i - X^T y) \right) \right)$$

Q8.

Modify the function compute square loss, to compute J(θ) for a given θ. You might want to create a small dataset for which you can compute J(θ) by hand, and verify that your compute square loss function returns the correct value.

```
#####################################
### The square loss function
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
    """
    #TODO
    # Recall that formula  J(θ) is (1/m) * (‖Xb - y‖2)^2, where m = X.shape[0]
    loss = (1 / X.shape[0]) * (np.linalg.norm((X @ theta) - y) ** 2)
    return loss
```

```
# Verify compute_square_loss function
theta = np.zeros(X_train.shape[1], )
compute_square_loss(X_train, y_train, theta)
```

```
7.961518343622414
```

Q9.

Modify the function compute square loss gradient, to compute ∇θJ(θ). You may again want to use a small dataset to verify that your compute square loss gradient function returns the correct value.

```
#####################################
### The gradient of the square loss function
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
    # Recall that formula is (2/m) * (X.T @ X @ theta - X.T @ y), where m = X.shape[0]
    grad = (2 / X.shape[0]) * ((X.T @ X @ theta) - (X.T @ y))
    return grad
```

```
# Verify compute_square_loss_gradient function
theta = np.zeros(X_train.shape[1], )
compute_square_loss_gradient(X_train, y_train, theta)
```

```
array([ 0.25182065, -0.15015092, -0.40509526, -0.48635482, -0.09047498,
       -0.14102752, -0.19706508, -0.19706508, -0.43171119, -0.64257159,
       -0.69600073, -0.37699938, -0.00881756,  0.1961221 , -0.61472345,
       -0.71499271, -0.57656883, -0.11765843, -0.16572975, -0.16572975,
       -0.16572975, -0.2405182 , -0.2405182 , -0.2405182 , -0.25532376,
       -0.25532376, -0.25532376, -0.26195435, -0.26195435, -0.26195435,
       -0.2655612 , -0.2655612 , -0.2655612 , -0.29115783, -0.29115783,
       -0.29115783, -0.32684613, -0.32684613, -0.32684613, -0.30492033,
       -0.30492033, -0.30492033, -0.29459583, -0.29459583, -0.29459583,
       -0.28881282, -0.28881282, -0.28881282,  0.36972349])
```

Q10.

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 so which can work for any objective function.

```
#####################################
### Gradient checker
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
    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
    # create an d*d array with ones on the diagonal and zeros elsewhere, where d = X.shape[1]
    e_i = np.eye(X.shape[1])

    # update approx_grad with formula: (J(theta + epsilon * e_i) - J(theta - epsilon * e_i)) /(2*epsilon)
    for i in range(X.shape[1]):
        j_plus = compute_square_loss(X, y, theta + (epsilon * e_i[i]))
        j_minus = compute_square_loss(X, y, theta - (epsilon * e_i[i]))
        approx_grad[i] = (j_plus - j_minus)/(2 * epsilon)

    # Euclidean distance
    euc_dist = np.linalg.norm(approx_grad - true_gradient)

    # If the Euclidean distance exceeds tolerance, we say the gradient is incorrect.
    if euc_dist > tolerance:
        return False

    # If not, return True
    return True
```

```
# test
theta = np.zeros(49,)
grad_checker(X_train, y_train, theta, epsilon=0.01, tolerance=1e-4)
```

```
True
```

Q11.

Complete batch gradient descent. Note the phrase batch gradient descent distinguishes between stochastic gradient descent or more generally minibatch gradient descent.

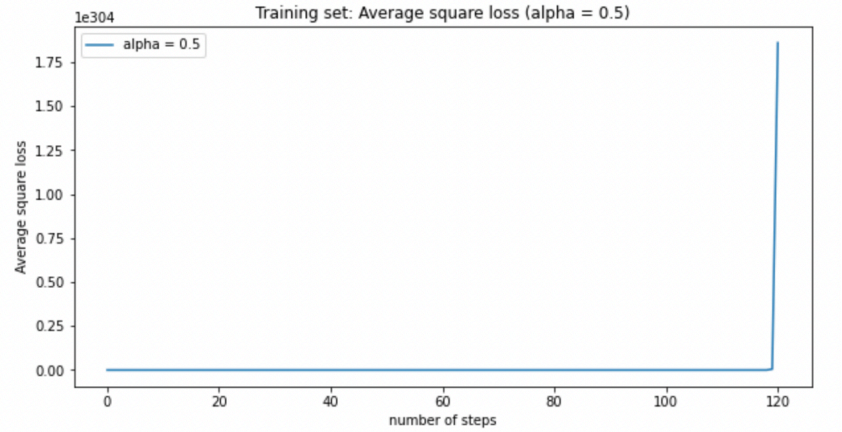
```
#####################################
### Batch gradient descent
def batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=False):

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

    #TODO
    # update theta_hist and loss_hist
    for i in range(num_step+1):
        theta_hist[i, :] = theta
        loss_hist[i] = compute_square_loss(X, y, theta)

        theta -= alpha * compute_square_loss_gradient(X, y, theta)

        # if grad_check is still False, then break the loop and return result
        grad_check = grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4)

        if grad_check == True:
            assert(grad_checker)


    return theta_hist, loss_hist
```

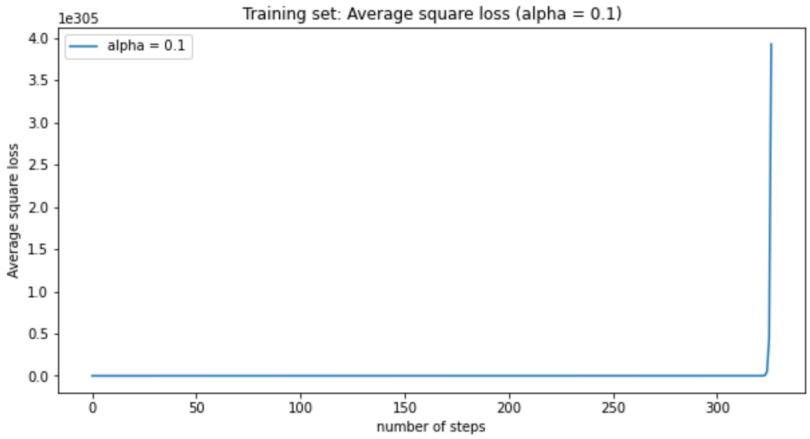
Q12.

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.

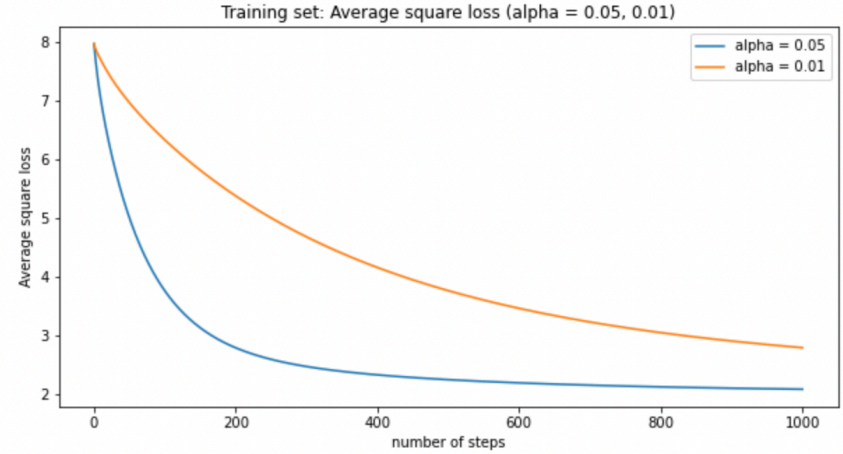
Ans)
For large step sizes (alpha = 0.5, 0.1), loss doesn't seem to converge.
However, as we use small step sizes(alpha = 0.05, 0.01), it converges.

Also, as we use more small step sizes(alpha = 0.01), it converges more slowly.
In plot, we can see that alpha = 0.05 converges most quickly among other alphas






```
plt.figure(figsize= (10,5))
alpha = 0.05
theta_hist_3, loss_hist_3 = batch_grad_descent(X_train, y_train, alpha=alpha, num_step=1000, grad_check=False)
plt.plot(np.arange(loss_hist_3.shape[0]), loss_hist_3, label = 'alpha = {}'.format(alpha))

alpha = 0.01
theta_hist_4, loss_hist_4 = batch_grad_descent(X_train, y_train, alpha=alpha, num_step=1000, grad_check=False)
plt.plot(np.arange(loss_hist_4.shape[0]), loss_hist_4, label = 'alpha = {}'.format(alpha))

plt.title('Training set: Average square loss (alpha = 0.05, 0.01)')
plt.xlabel('number of steps')
plt.ylabel('Average square loss')
plt.legend()
plt.show()
```

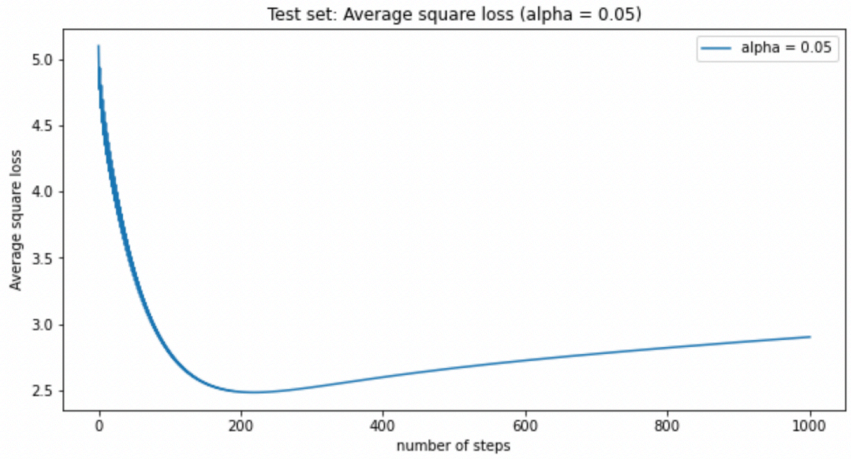


Q13.

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.

Ans) For learning rate: 0.05, test error first decreases and then increases. (Overfitting)

```
# alpha = 0.05
num_step = 1000
loss_hist_test_3 = np.zeros(num_step + 1)  #Initialize loss_hist
for i in range(num_step+1):
    loss_hist_test_3[i] = compute_square_loss(X_test, y_test, theta_hist_3[i])

plt.figure(figsize= (10,5))
alpha = 0.05
plt.plot(np.arange(loss_hist_test_3.shape[0]), loss_hist_test_3, label = 'alpha = {}'.format(alpha))
plt.title('Test set: Average square loss (alpha = 0.05)')
plt.xlabel('number of steps')
plt.ylabel('Average square loss')
plt.legend()
plt.show()
```



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

We can apply Q5, Q6 calculation,

$J_\lambda(\theta) = \frac{1}{m}\|X\theta - y\|_2^2 + \lambda\theta^T\theta$,

- $\nabla_\theta J_\lambda(\theta) = \frac{1}{m}\nabla_\theta\|X\theta - y\|_2^2 + \nabla_\theta\lambda\theta^T\theta$

$= \frac{2}{m}(X^TX\theta - X^Ty) + 2\lambda\theta$ (Q6 calculation applied)

- Gradient descent algorithm

Repeat $\theta \leftarrow \theta - \eta\nabla_\theta J(\theta)$, where $\nabla_\theta J(\theta) = \frac{2}{m}(X^TX\theta - X^Ty) + 2\lambda\theta$

$(\Rightarrow \theta_{i+1} = \theta_i - \eta(\frac{2}{m}(X^TX\theta_i - X^Ty) + 2\lambda\theta_i)$

Q15.

Implement compute regularized square loss gradient.

```
#######################################
### The gradient of regularized batch gradient descent
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    #TODO
    # Recall that formula is (2/m) * (X.T @ X @ theta - X.T @ y) + 2*lambda*theta, where m = X.shape[0]
    grad = (2 / X.shape[0]) * ((X.T @ X @ theta) - (X.T @ y)) + (2 * lambda_reg * theta)
    return grad
```

Q16.

Implement regularized grad descent.

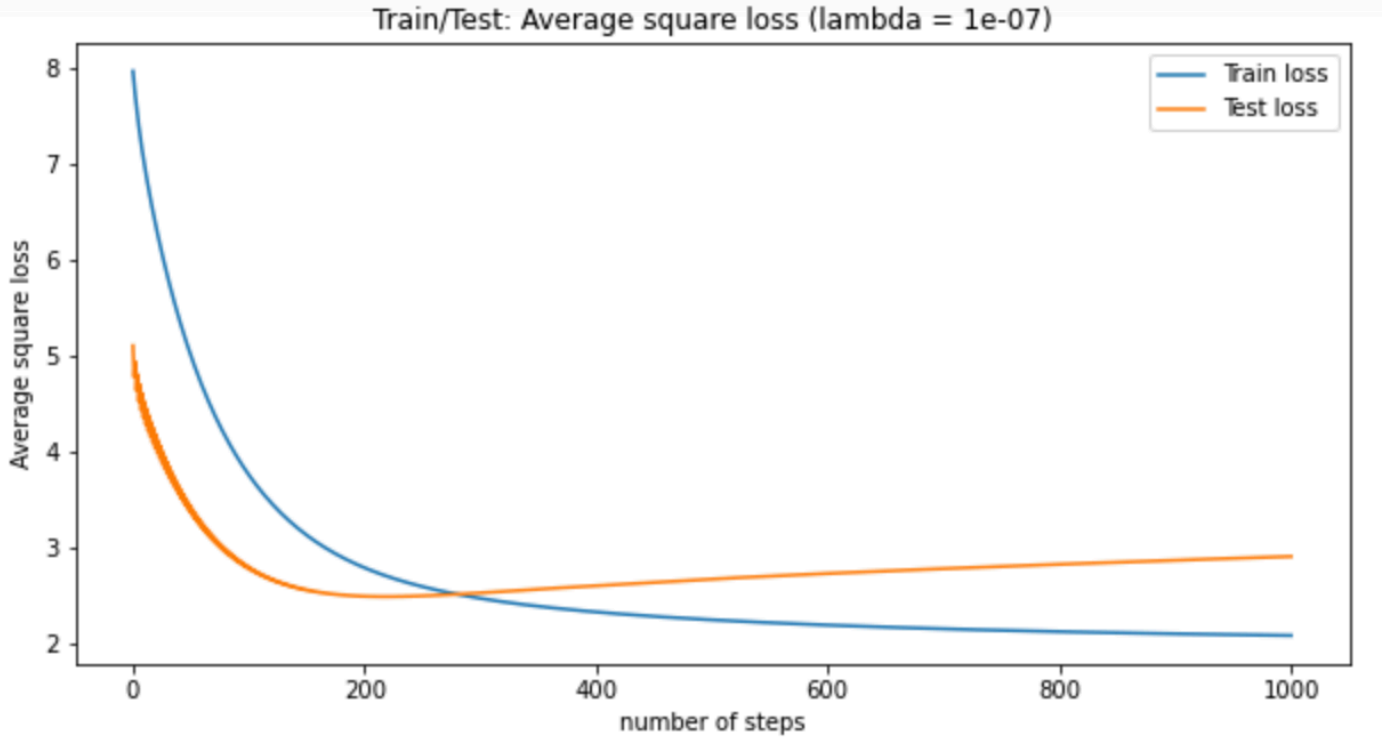
```
#######################################
### Regularized batch gradient descent
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2, num_step=1000):
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.zeros(num_features) #Initialize theta
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist

    #TODO
    # update theta_hist and loss_hist
    for i in range(num_step+1):
        theta_hist[i, :] = theta
        loss_hist[i] = compute_square_loss(X, y, theta)

        theta -= alpha * compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)

    return theta_hist, loss_hist
```

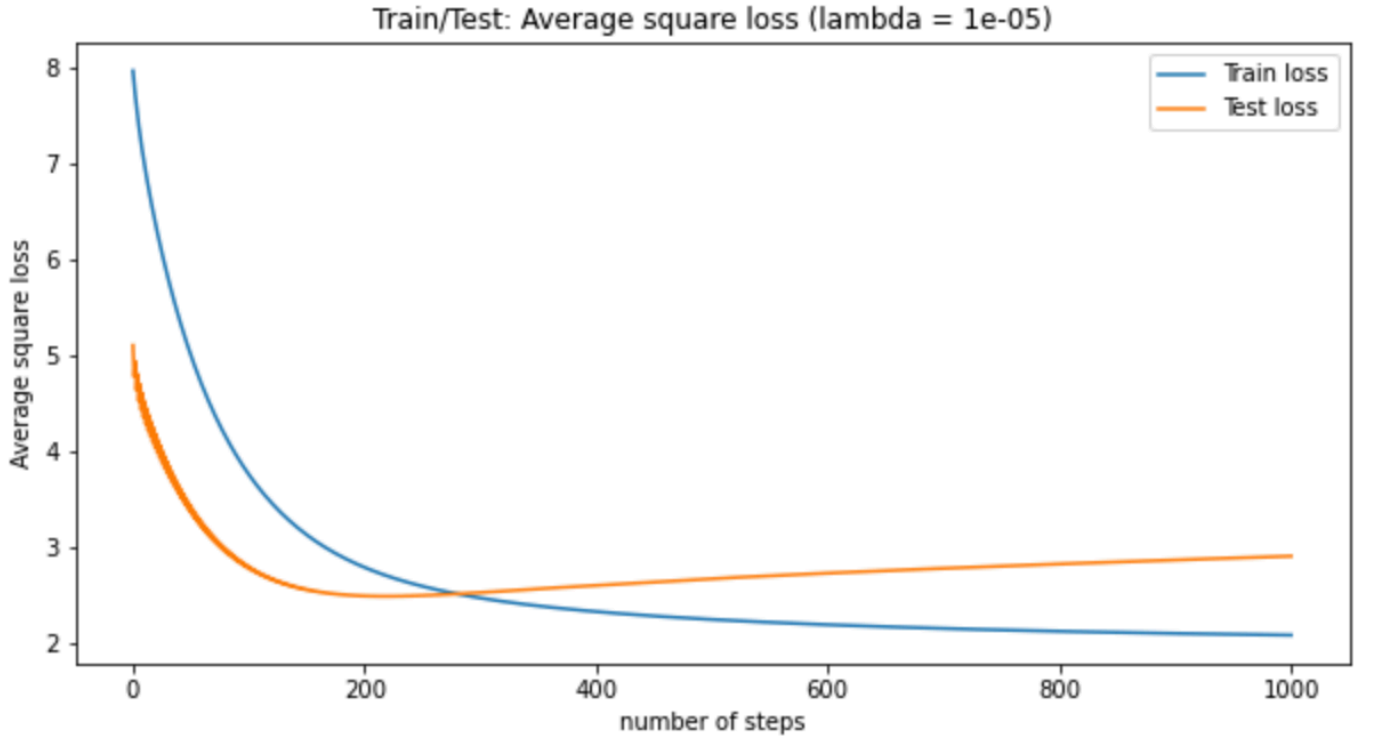
Q17.

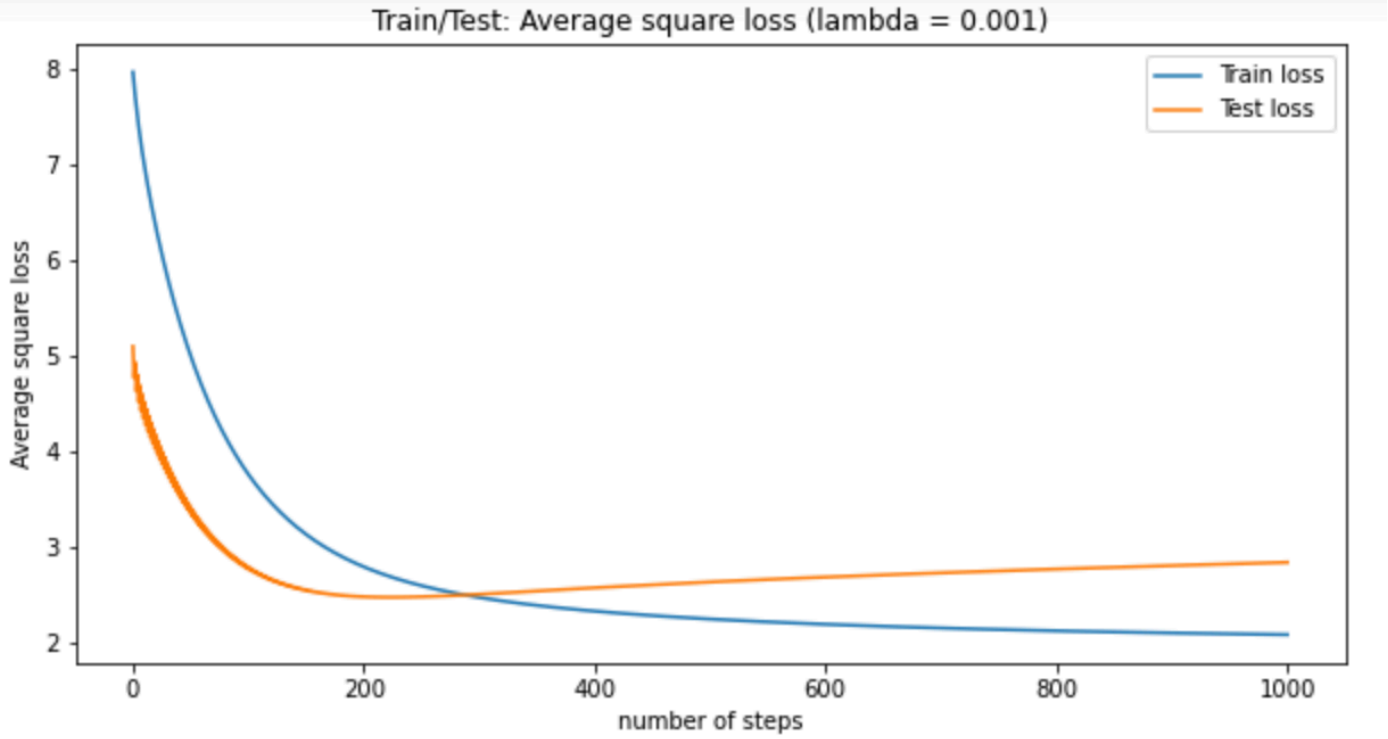
Choosing a reasonable step-size, plot training average square loss and the test average square loss (just the average square loss part, without the regularization, in each case) as a function of the training iterations for various values of λ. What do you notice in terms of overfitting?

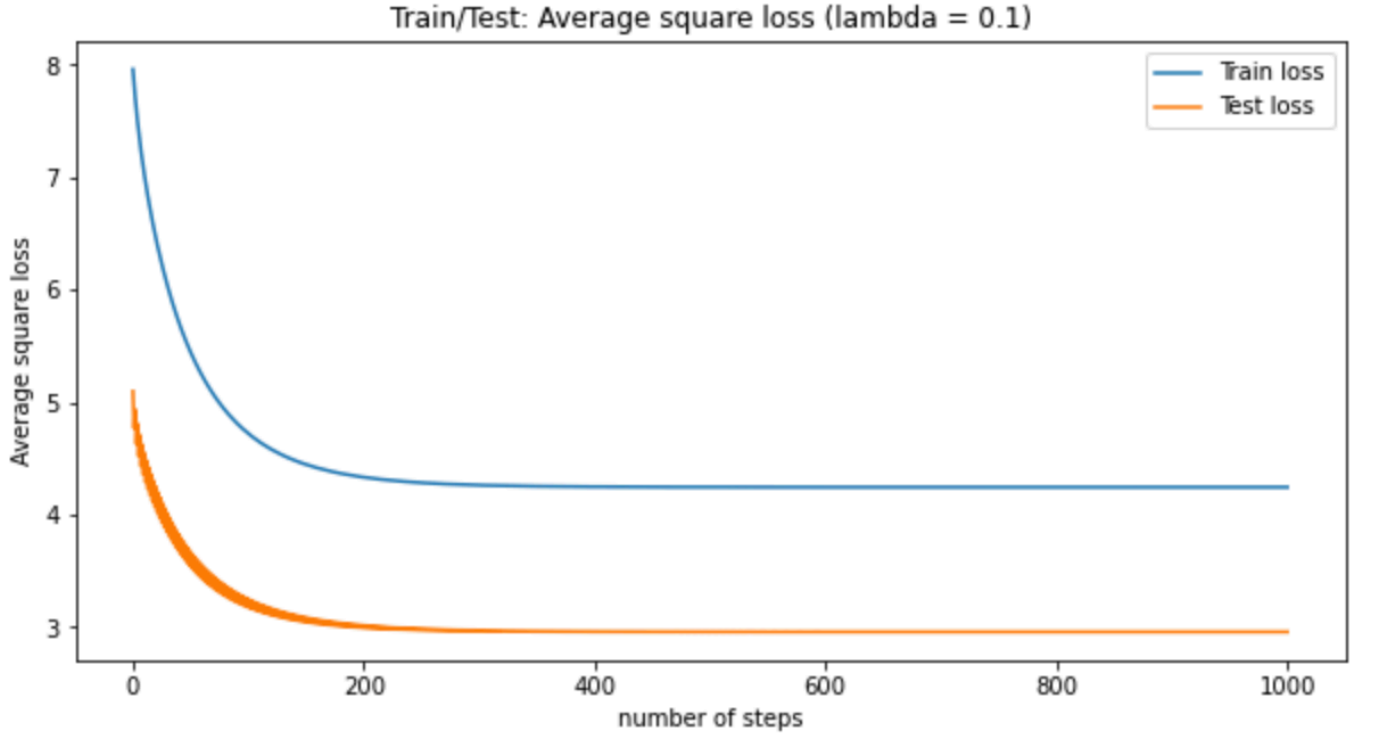
Ans) As we select some reasonable step-size such as lambda = 10-7, 10-5, 10-3, we can see that overfitting occurs for given models.**
If we try to minimize train loss, then the model will overfit to the training set and it fails to predict test dataset, resulting increase in train loss.

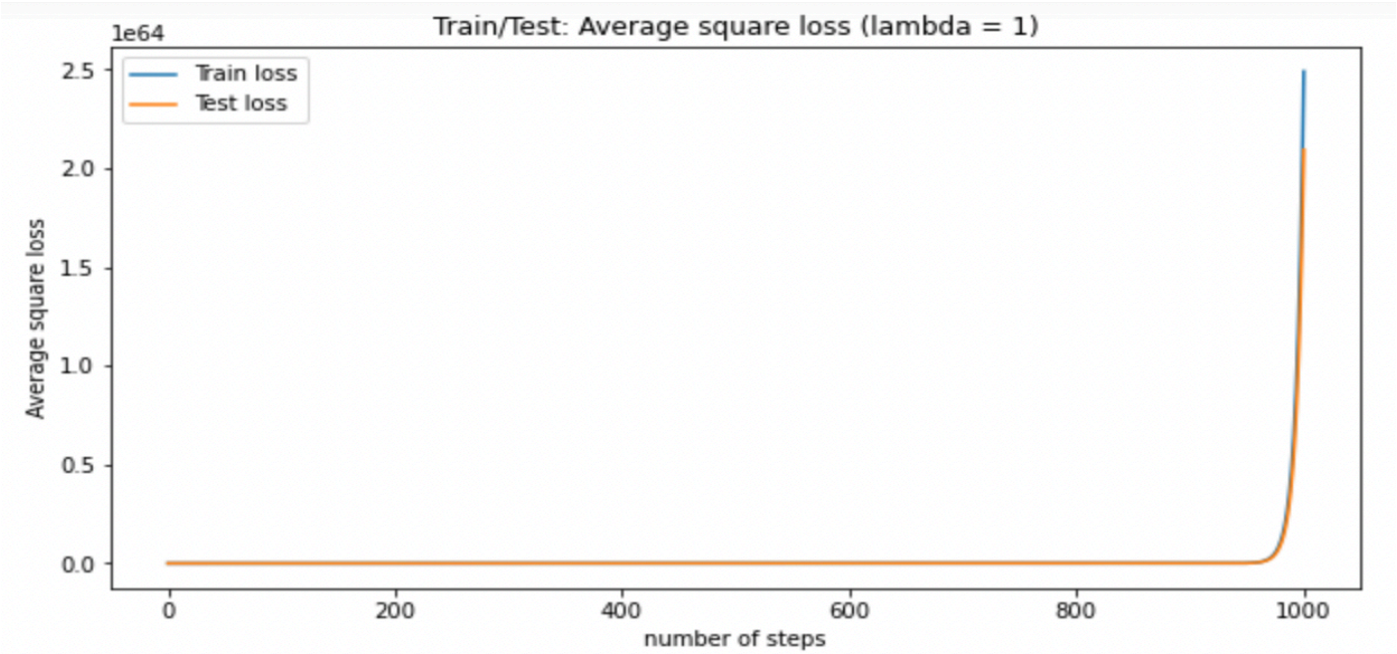
```
lambda_lists = [10**-7, 10**-5, 10**-3, 10**-1, 1, 10, 100]
alpha = 0.05 # step-size

for lambda_list in lambda_lists:
    plt.figure(figsize= (10,5))
    # train set
    theta_hist_train, loss_hist_train = regularized_grad_descent(X_train, y_train, alpha=0.05, \
                                                                 lambda_reg=lambda_list, num_step=1000)

    plt.plot(np.arange(loss_hist_train.shape[0]), loss_hist_train, label = 'Train loss')

    # test set
    num_step=1000
    loss_hist_test = np.zeros(num_step + 1)  #Initialize loss_hist
    for i in range(num_step+1):
        loss_hist_test[i] = compute_square_loss(X_test, y_test, theta_hist_train[i])

    plt.plot(np.arange(loss_hist_test.shape[0]), loss_hist_test, label = 'Test loss')

    plt.title('Train/Test: Average square loss (lambda = {})'.format(lambda_list))
    plt.xlabel('number of steps')
    plt.ylabel('Average square loss')
    plt.legend()
    plt.show()
```


Train/Test: Average square loss (lambda = 1e-07)
Train loss
Test loss
Average square loss
number of steps
0
200
400
600
800
1000
2
3
4
5
6
7
8

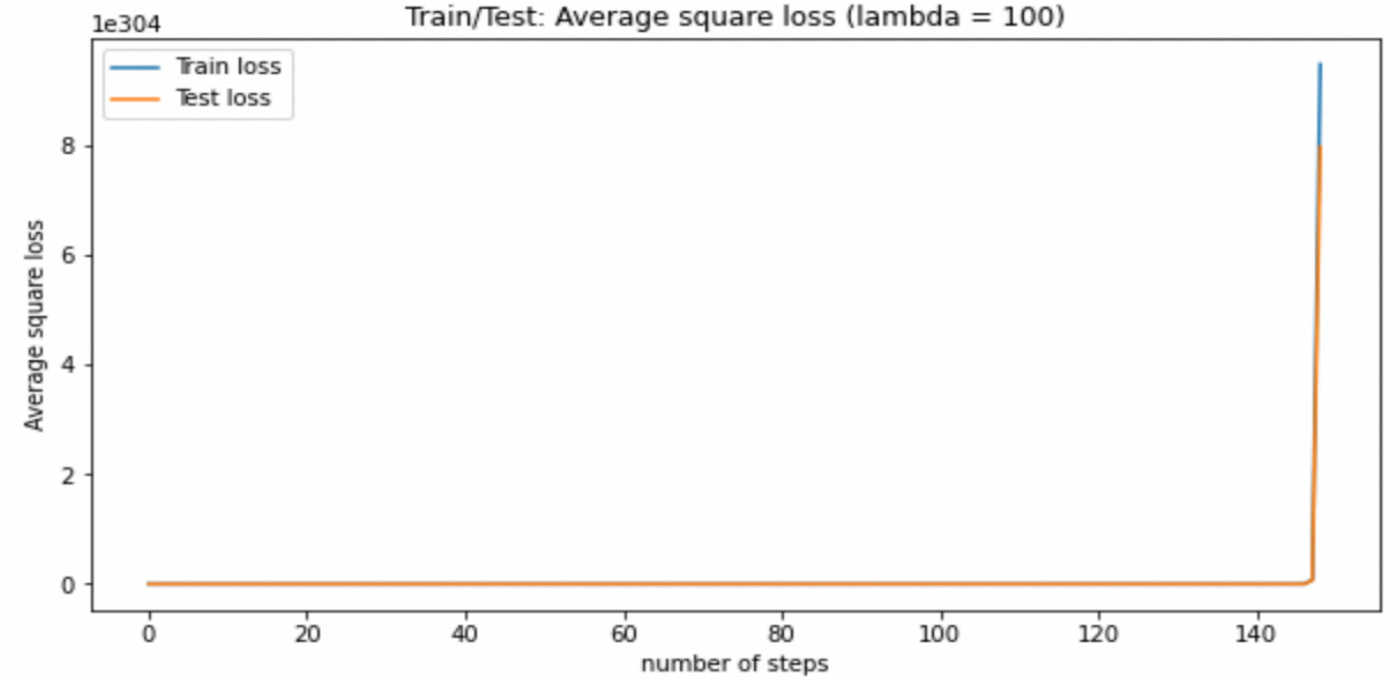

Train/Test: Average square loss (lambda = 1e-05)
Train loss
Test loss
Average square loss
number of steps
0
200
400
600
800
1000
2
3
4
5
6
7
8


Train/Test: Average square loss (lambda = 0.001)
Train loss
Test loss
Average square loss
number of steps
0
200
400
600
800
1000
2
3
4
5
6
7
8


Train/Test: Average square loss (lambda = 0.1)
Train loss
Test loss
Average square loss
number of steps
0
200
400
600
800
1000
3
4
5
6
7
8


Train/Test: Average square loss (lambda = 1)
1e64
Train loss
Test loss
Average square loss
number of steps
0.0
0.5
1.0
1.5
2.0
2.5
0
200
400
600
800
1000


Train/Test: Average square loss (lambda = 10)
1e305
Train loss
Test loss
Average square loss
number of steps
0
1
2
3
4
5
6
7
0
100
200
300
400
500


Train/Test: Average square loss (lambda = 100)
1e304
Train loss
Test loss
Average square loss
number of steps
0
2
4
6
8
0
20
40
60
80
100
120
140

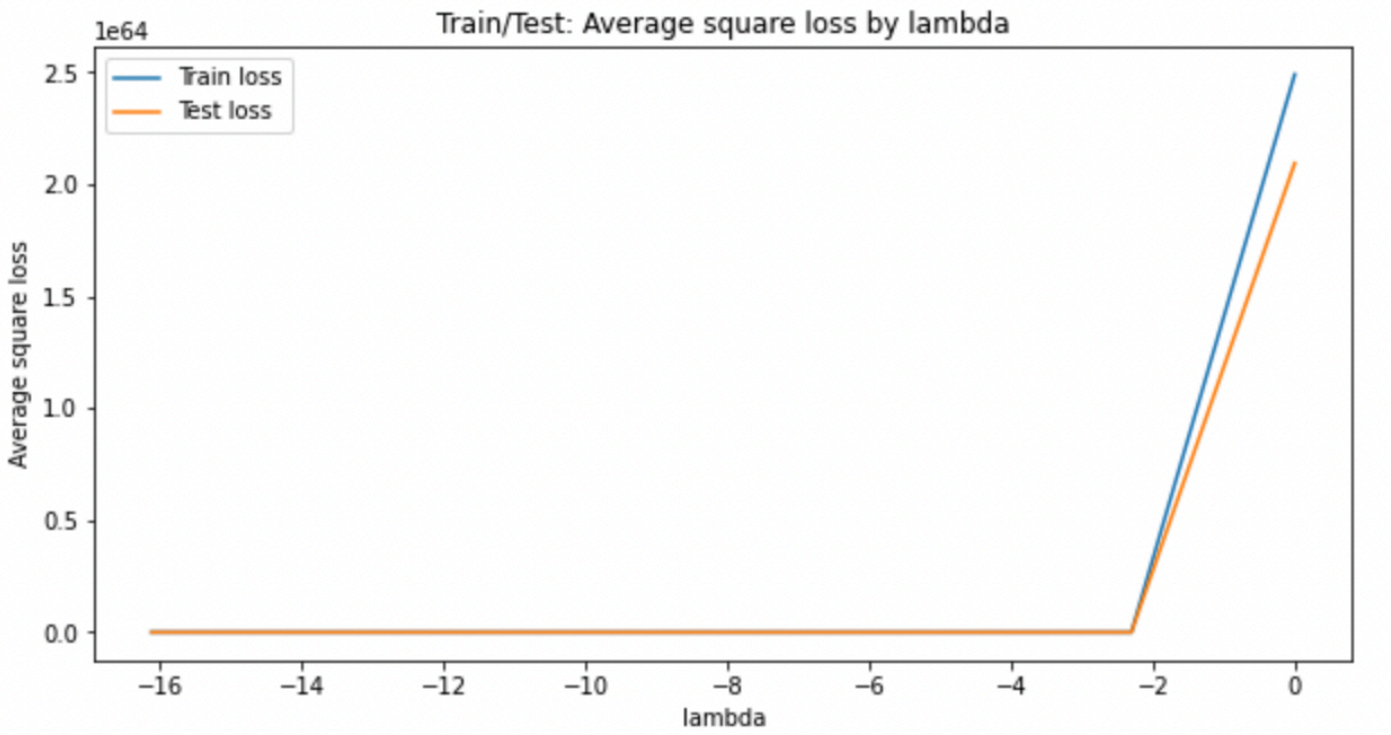
Q18.

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(λ) on the x-axis rather than λ. Which value of λ would you choose ?

Ans) Select 10-3, as it shows the lowest test loss(=2.845643139725183)**

```
lambda_lists = [10**-7, 10**-5, 10**-3, 10**-1, 1, 10, 100]
alpha = 0.05 # step-size

train_loss_by_lambda = []
test_loss_by_lambda = []

for lambda_list in lambda_lists:
    # train set
    theta_hist_train, loss_hist_train = regularized_grad_descent(X_train, y_train, alpha=alpha, \
                                                                 lambda_reg=lambda_list, num_step=1000)
    # Add the end of training result
    train_loss_by_lambda.append(loss_hist_train[-1])

    # test set, calculate loss by using end of training theta
    test_loss_by_lambda.append(compute_square_loss(X_test, y_test, theta_hist_train[-1]))

print('train: ', train_loss_by_lambda)
print()
print('test: ', test_loss_by_lambda)

plt.figure(figsize = (10,5))
plt.plot(np.log(lambda_lists), train_loss_by_lambda, label = 'Train loss')
plt.plot(np.log(lambda_lists), test_loss_by_lambda, label = 'Test loss')

plt.title('Train/Test: Average square loss by lambda')
plt.xlabel('lambda')
plt.ylabel('Average square loss')
plt.legend()
plt.show()
```

```
train:  [2.077700614426301, 2.0778239254952293, 2.0913500007595975, 4.245811582850459, 2.490766500640904e+64, inf,
nan]

test:  [2.9013357289427684, 2.900754508857025, 2.845643139725183, 2.9599402234396774, 2.0942512356184937e+64, inf,
nan]
```



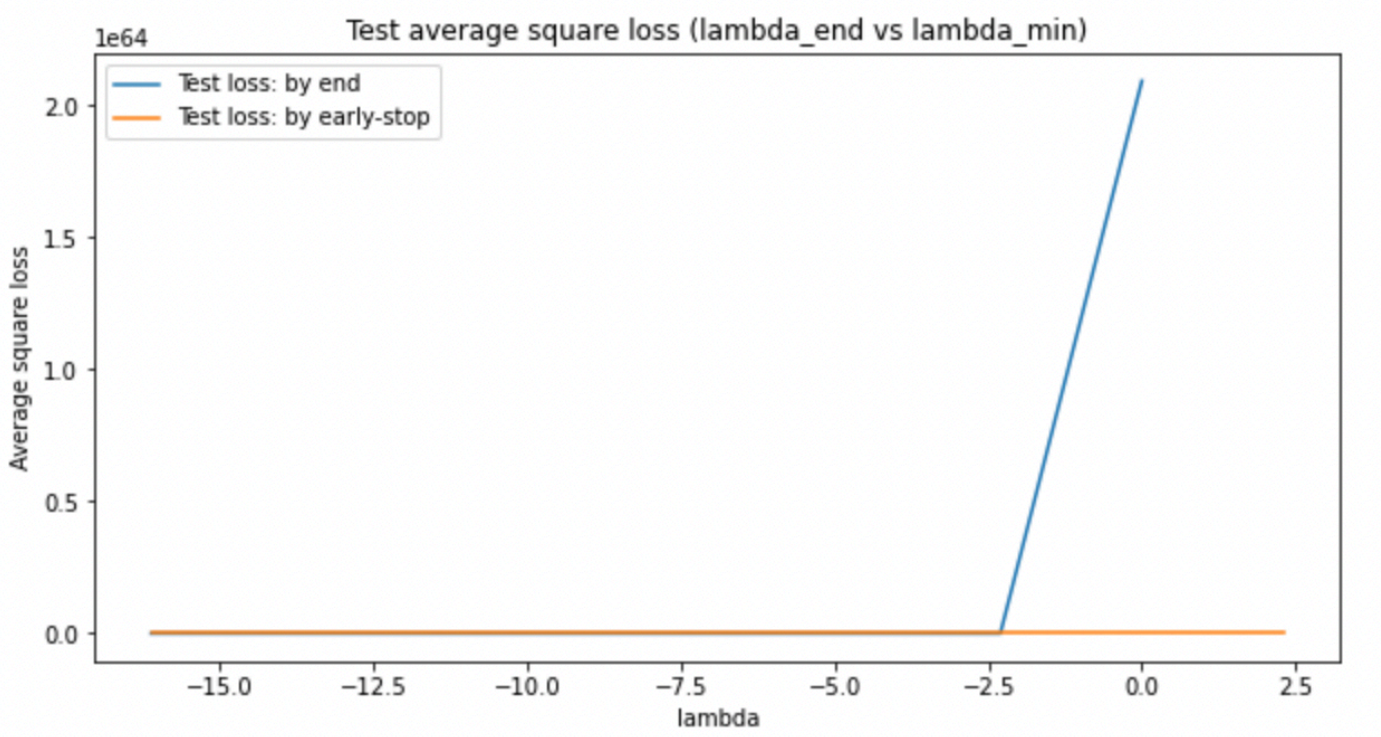
Q19.

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?

Ans) In this step-size: 0.05, result is same and therefore select same λ = 10 **(-3)

```
lambda_lists = [10**-7, 10**-5, 10**-3, 10**-1, 1, 10, 100]
alpha = 0.05 # step-size
test_loss_by_lambda_end = []
test_loss_by_lambda_min = []

for lambda_list in lambda_lists:
    # this part is same as Q18, to calculate test loss by using last index
    # train dataset
    theta_hist_train, loss_hist_train = regularized_grad_descent(X_train, y_train, alpha=alpha, \
                                                                 lambda_reg=lambda_list, num_step=1000)

    # test set, calculate loss by using end of training theta
    test_loss_by_lambda_end.append(compute_square_loss(X_test, y_test, theta_hist_train[-1]))


    # Assuming that minimum train error index is the early stop point,
    # apply that index of train theta to test set
    min_idx_train = np.argmin(loss_hist_train)
    test_loss_by_lambda_min.append(compute_square_loss(X_test, y_test, theta_hist_train[min_idx_train]))

print('end: ', test_loss_by_lambda_end)
print()
print('min: ', test_loss_by_lambda_min)

plt.figure(figsize = (10,5))
plt.plot(np.log(lambda_lists), test_loss_by_lambda_end, label = 'Test loss: by end')
plt.plot(np.log(lambda_lists), test_loss_by_lambda_min, label = 'Test loss: by early-stop')

plt.title('Test average square loss (lambda_end vs lambda_min)')
plt.xlabel('lambda')
plt.ylabel('Average square loss')
plt.legend()
plt.show()
```

```
end:  [2.9013357289427684, 2.900754508857025, 2.845643139725183, 2.9599402234396774, 2.0942512356184937e+64, inf, nan]

min:  [2.9013357289427684, 2.900754508857025, 2.845643139725183, 2.9599402234396774, 4.395676232924864, 4.7713708963361325, nan]
```

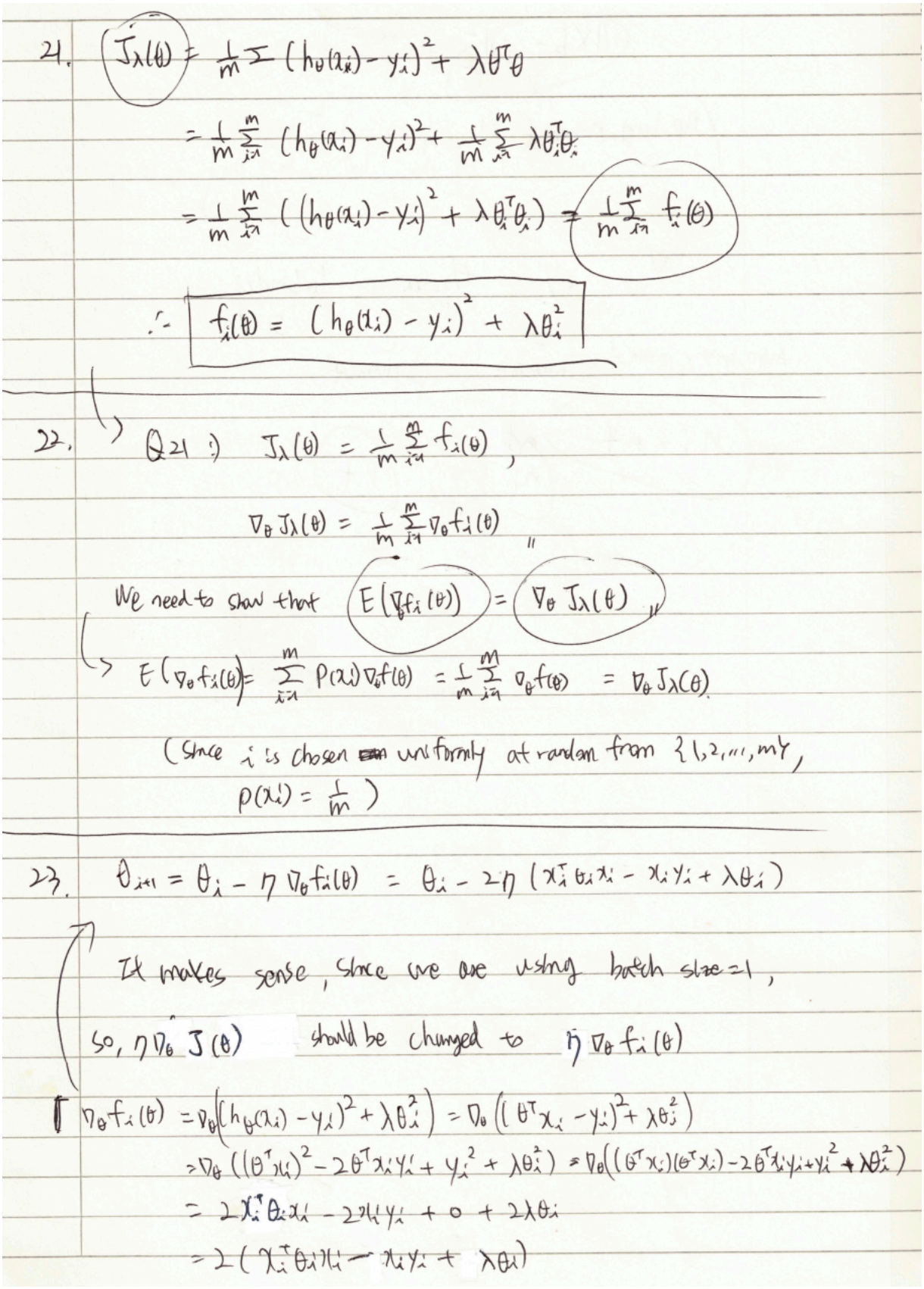


Q20.

What λ would you select in practice and why?

Ans) Select λ that best minimizes test error. In practice, we divide train/test dataset, and our goal is to make a model that can predict well for new data. So, it is reasonable to select λ that minimizes test error. We shouldn't select λ that minimizes train error, as that model can overfit to train data and does not perform well in test dataset.

Also, we can think of some early stopping point when deciding test error for each given λ. We might not able to check entire dataset so making an early stop point when test error doesn't minimizes for some period of iteration(ex. early stop when you find that iteration doesn't decreases test error for next 50 times), or select among some given iteration (ex. early stop when you run model 200 times and find minimum test error)



Optional) Q24. Implement stochastic grad descent.

```
### Stochastic gradient descent: C = 0.1
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2, num_epoch=1000, eta0=False):
    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
    # initialize step_size as alpha
    step_size = alpha
    C = 0.1

    for epoch in range(num_epoch):
        # check alpha and confirm final step_size
        if type(alpha) == float:
            step_size = alpha
        elif alpha == '1/sqrt(t)':
            step_size = C * 1/np.sqrt(epoch+1)
        elif alpha == '1/t':
            step_size = C * (1/(epoch+1))

        # shuffle(randomize) index
        index = np.array(range(num_instances))
        np.random.shuffle(index)

        for j in index:
            # calculate theta and loss
            theta_hist[epoch, j, :] = theta
            loss = compute_square_loss(X[j, :].reshape(1,X[j, :].shape[0]), y[j].reshape(-1,), theta)
            loss_hist[epoch, j] = loss

            # update theta
            gradient = compute_regularized_square_loss_gradient(X[j, :].reshape(1,X[j, :].shape[0]), \
                                                                y[j].reshape(-1,), theta, lambda_reg)
            theta = theta - step_size * gradient

    return theta_hist, loss_hist
```

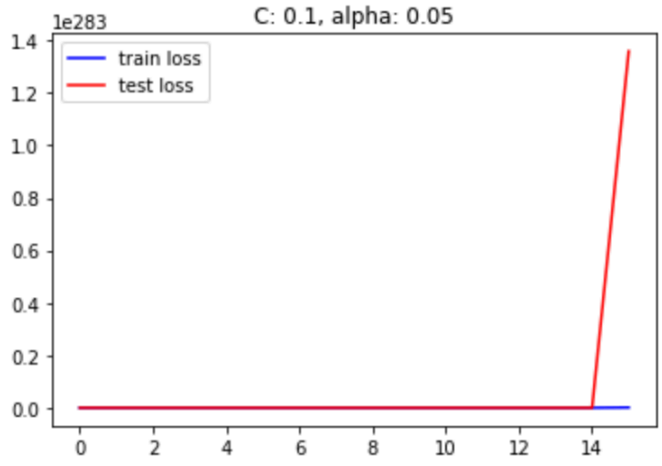
Q25. Use SGD to find $\theta\lambda*$ that minimizes the ridge regression objective for the λ you selected

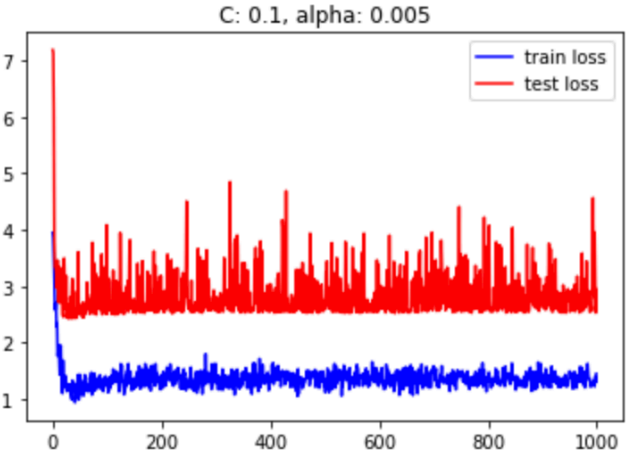
in the previous problem. (If you could not solve the previous problem, choose λ = 10–2). Try a few fixed step sizes (at least try ηt ∈ {0.05, .005}). Note that SGD may not converge with fixed step size. Simply note your results. Next try step sizes that decrease with the step number according to the following schedules: ηt = C and ηt = √C , C ≤ 1. Please tt 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?

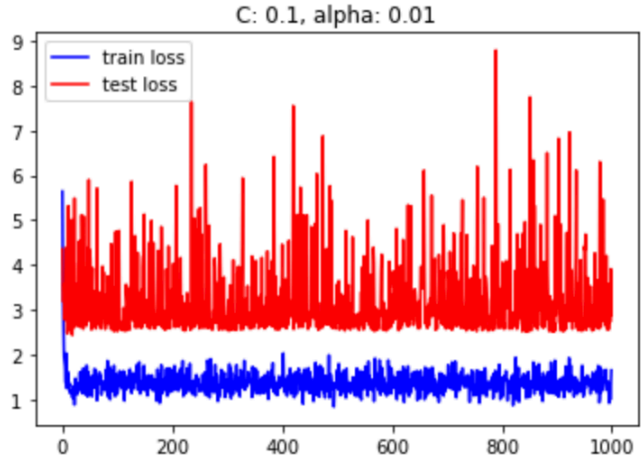
Ans) For large step size(alpha = 0.05), the model may diverge. As we decrease step size, we can see the trend of train/test loss converging.

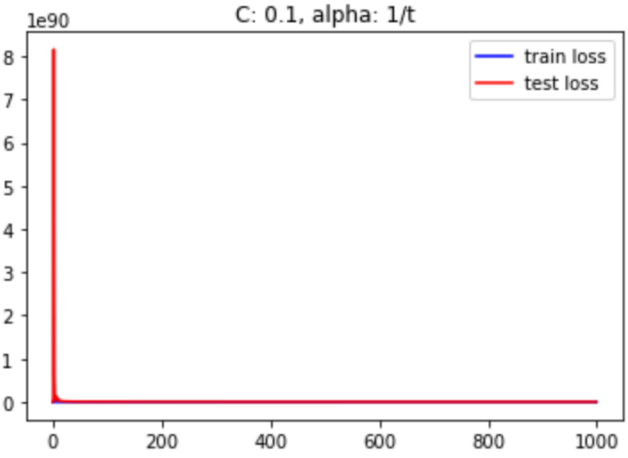
Also, when we change C from 0.1 to lower value, it means that for given alpha = 1/t or 1/sqrt(t), it makes step sizes more smaller which may lead to slower convergence.

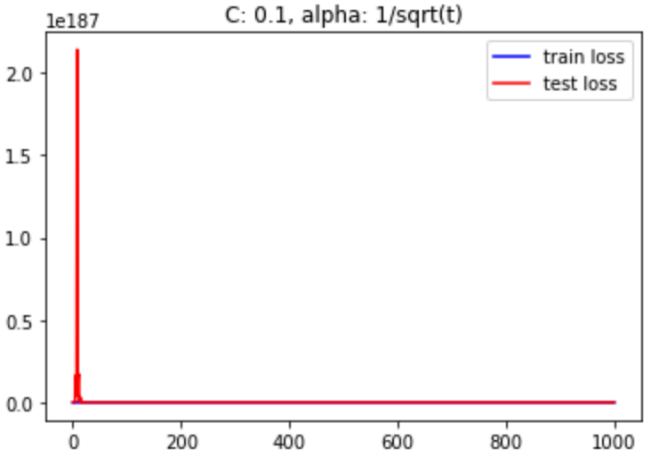
```
alphas = [0.05, 0.01, 0.005, '1/t', '1/sqrt(t)']

for alpha in alphas:
    theta_hist, loss_hist = stochastic_grad_descent(X_train, y_train, alpha=alpha, lambda_reg=10**-2, num_epoch=1000

    num_epoch = 1000

    train_loss_hist = np.zeros(num_epoch)
    test_loss_hist = np.zeros(num_epoch)

    loss_hist = loss_hist[:, -1]

    for step in range(num_epoch):
        test_loss_hist[step] = compute_square_loss(X_test, y_test, theta_hist[step, 1, :])

    plt.title('C: 0.1, alpha: {}'.format(alpha))
    plt.plot(np.arange(num_epoch), loss_hist, color = 'blue', label = 'train loss')
    plt.plot(np.arange(num_epoch), test_loss_hist, color = 'red', label = 'test loss')
    plt.legend()
    plt.show()
```

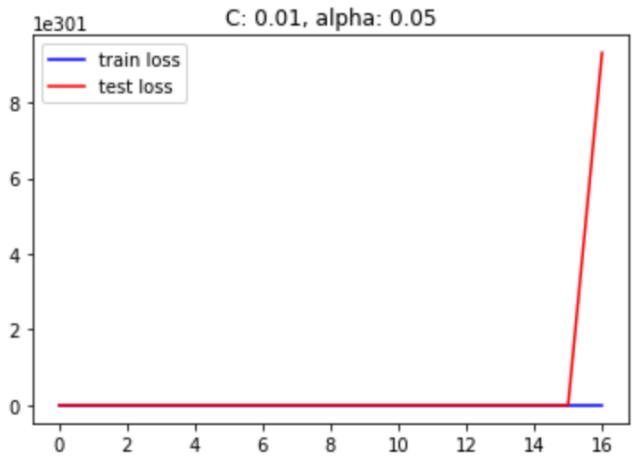

C: 0.1, alpha: 0.05
1e283
train loss
test loss

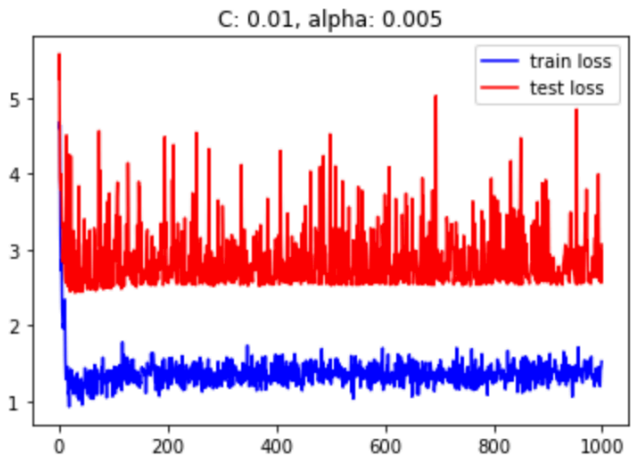

C: 0.1, alpha: 0.005
train loss
test loss

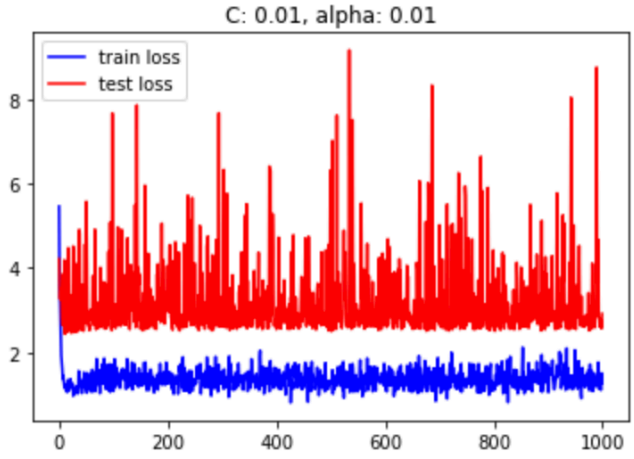

C: 0.1, alpha: 0.01
train loss
test loss

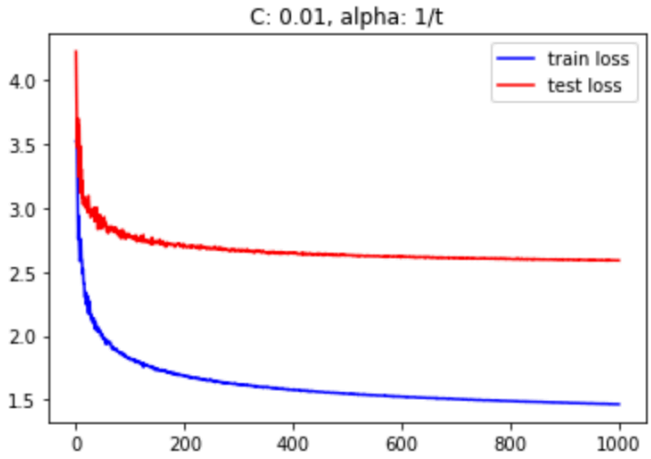

C: 0.1, alpha: 1/t
1e90
train loss
test loss

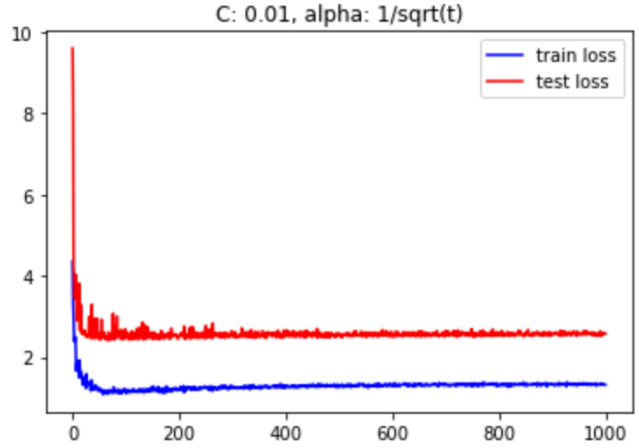

C: 0.1, alpha: 1/sqrt(t)
1e187
train loss
test loss

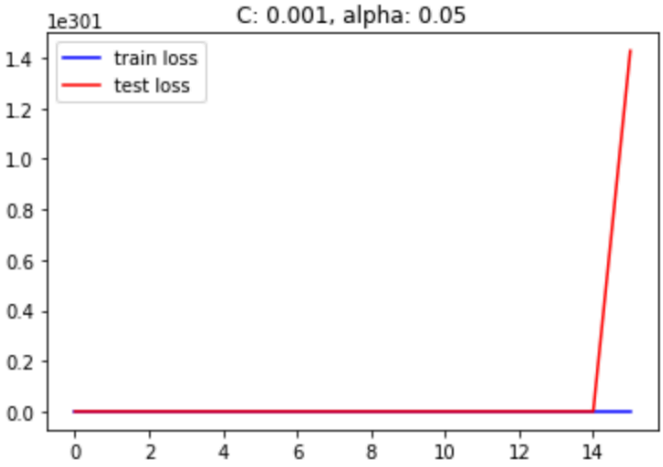

C: 0.01, alpha: 0.05
1e301
train loss
test loss

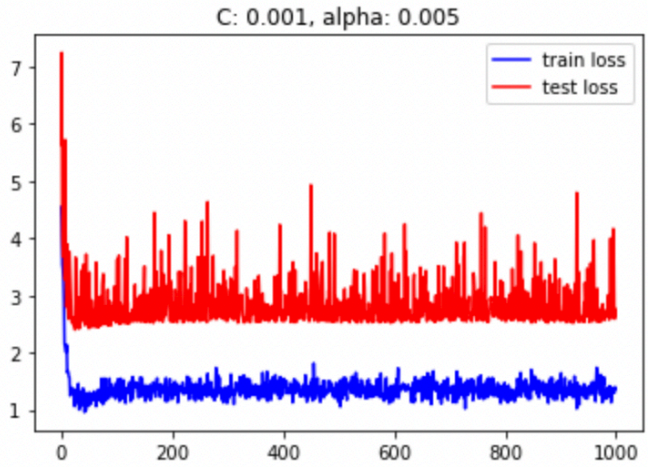

C: 0.01, alpha: 0.005
train loss
test loss

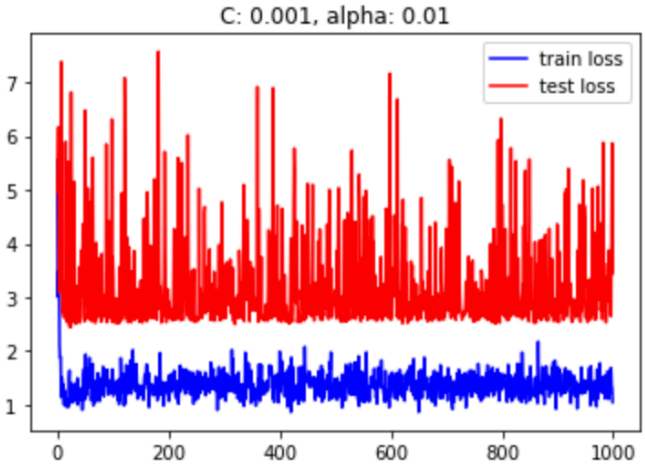

C: 0.01, alpha: 0.01
train loss
test loss

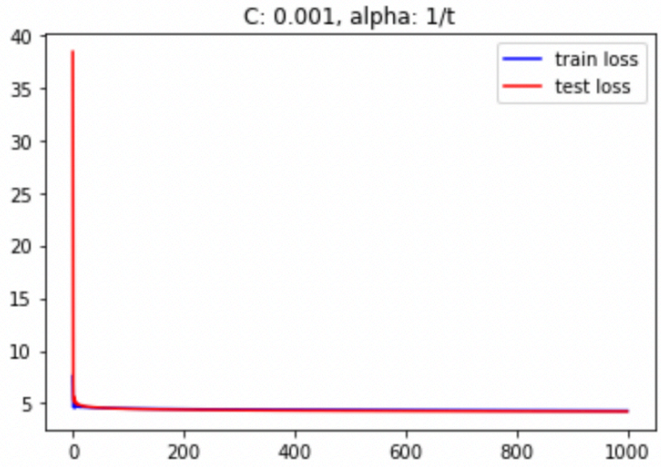

C: 0.01, alpha: 1/t
train loss
test loss

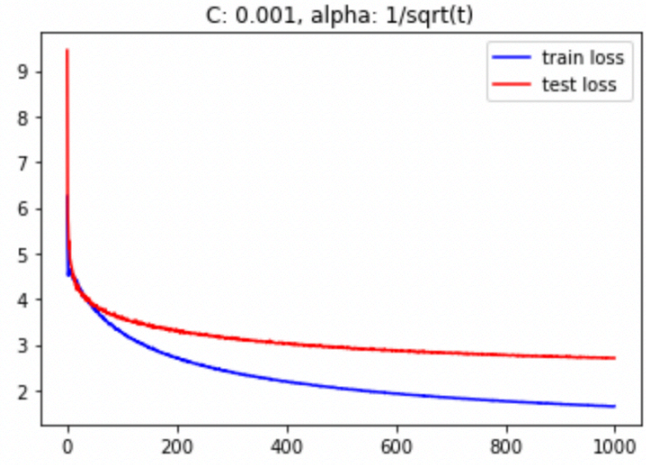

C: 0.01, alpha: 1/sqrt(t)
train loss
test loss


C: 0.001, alpha: 0.05
1e301
train loss
test loss


C: 0.001, alpha: 0.005
train loss
test loss


C: 0.001, alpha: 0.01
train loss
test loss


C: 0.001, alpha: 1/t
train loss
test loss


C: 0.001, alpha: 1/sqrt(t)
train loss
test loss

Q 26. Logistic / Log loss : $\ell(m) = \log(1+e^{-m})$

$m = y h_{\theta,b}(x)$, $y_i \in \{-1, 1\}$

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

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

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\frac{1+y_i}{2}\left(\log\left(1+e^{-y_i h_{\theta,b}(x_i)}\right)\right) + \frac{1-y_i}{2}\left(\log\left(1+e^{-y_i h_{\theta,b}(x_i)}\right)\right)\right]$$

- if $y_i = 1$, $1+y_i = 2$, $1-y_i = 0$, $\ell(m) = \ell(y_i h_{\theta,b}(x_i)) = \log(1+e^{-h_{\theta,b}(x_i)})$

if $y_i = -1$, $1+y_i = 0$, $1-y_i = 2$, $\ell(m) = \ell(y_i h_{\theta,b}(x_i)) = \log(1+e^{h_{\theta,b}(x_i)})$

∴ We can say that if $y_i = 1$, we would be using first part of equation, and if $y_i = -1$, we would be using second part of equation

$$= \frac{1}{2m}\sum_{i=1}^{m}\left[(1+y_i)\log(1+e^{-h_{\theta,b}(x_i)}) + (1-y_i)\log(1+e^{h_{\theta,b}(x_i)})\right]$$

27. Adding l_1 penalty term to Q26 loss function

$$L(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left[(1+y_i)\log(1+e^{-h_{\theta,b}(x_i)}) + (1-y_i)\log(1+e^{h_{\theta,b}(x_i)})\right]$$

$$+ \alpha|\theta|$$

(where α is a regularization parameter)

Q28.

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).

```
# Q28 code
def classification_error(clf, X, y):
    ## TODO
    # initialize error result
    res = 0

    # predict
    y_pred = clf.predict(X)

    # check if prediction is equal to actual value
    for i in range(y_pred.shape[0]):
        if y_pred[i] != y[i]:
            res += 1

    return res / y_pred.shape[0]
```

```
# Checking with examples
X_train, X_test, y_train, y_test = pre_process_mnist_01()

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=0)

clf.fit(X_train, y_train)

test = classification_error(clf, X_test, y_test)
train = classification_error(clf, X_train, y_train)
print('train: ', train, end='\t')
print('test: ', test)
```

```
train:  0.0018178145829125 43    test:  0.0010250102501025 01
```

```
# You should check that your function returns the same value as 1 - clf.score(X, y).
# Checked that the result is same
alter_res = 1- clf.score(X_test, y_test)
print('Calculating by formula 1 - clf.score(X, y):{:.6f}'.format(alter_res))
```

```
Calculating by formula 1 - clf.score(X, y):0.001025
```

Q29.

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 the standard deviation as error bars.

```
# To speed up computations we will subsample the data.
# Using the function sub sample, restrict X train and y train to N train = 100
X_train, y_train = sub_sample(100, X_train, y_train)

# creating regularization parameters a (taking 10 values between 10-4 and 10-1))
alpha_list = [0.0001, 0.0005, 0.0007, 0.001, 0.005, 0.007, 0.01, 0.05, 0.07, 0.1]

# initialzing mean, std list
error_mean, error_std = [], []

# iterate by each alpha
for alpha in alpha_list:
    # we need 10 experiment, and the result will be saved in experiment_error list
    experiment_error = []

    # iterate 10 times
    for i in range(10):
        # create clf
        clf = SGDClassifier(loss='log', max_iter=1000,
                        tol=1e-3,
                        penalty='l1', alpha=alpha,
                        learning_rate='invscaling',
                        power_t=0.5,
                        eta0=0.01,
                        verbose=0)
        # fit
        clf.fit(X_train, y_train)

        # classification error
        experiment_error.append(classification_error(clf, X_test, y_test))

    # after 10 experiment, get mean and std
    error_mean.append(np.mean(experiment_error))
    error_std.append(np.std(experiment_error))
```

```
print('Mean: ', error_mean, '\n')
print('Std: ', error_std)
plt.figure(figsize = (10,5))
plt.errorbar(np.log(alpha_list), error_mean, yerr=error_std, ecolor = 'red', label = 'mean/std of test error')
# plt.errorbar(np.log(alpha_list), error_std, label = 'std of test error')

plt.title('Mean, std of test error by alphas')
plt.xlabel('alpha')
plt.ylabel('test error')
plt.legend()
plt.show()
```

```
Mean:  [0.005453054530545306, 0.005207052070520706, 0.005002050020500205, 0.005084050840508405, 0.005186551865518656, 0.005084050840508405, 0.005145551455514554, 0.005341533415334153, 0.0028290282902829027, 0.0021320213202132021]

Std:  [0.0006686964506068099, 0.000602580502529687, 0.0009085083971365451, 0.0006067506595714942, 0.0006677530740017356, 0.0008392574616834244, 0.0006702657943530879, 0.0003182487637609681, 0.0004656751411070335, 0.0002282806216822476 8]
```

Q30.

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

Ans) Randomness comes from sub-sampling train data(restrict X train and y train). Shuffling of the training set on random draws of the successive training points depending on what we implemented.

Also, randomness comes from parameter λ initialization for the gradient descent. (unless you don't use a random initialization but for instance a zero initialization)

Q31.

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

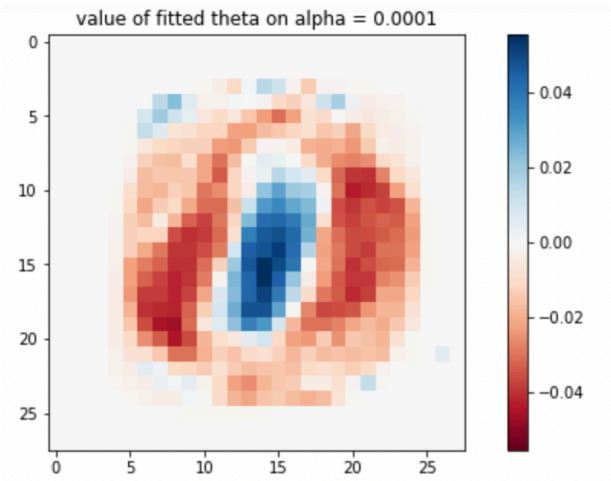
```
print('Optimal value a = {},\nby having both the lowest mean error of {:.4f} and std error of {:.4f}'.\
      format(alpha_list[np.argmin(error_mean)], np.min(error_mean), np.min(error_std)))
```

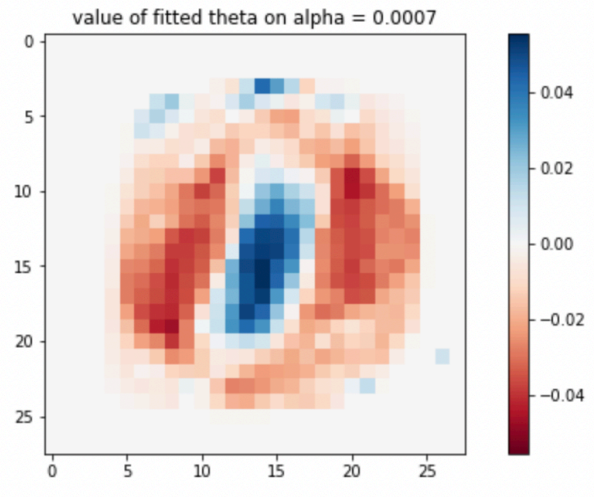
```
Optimal value a = 0.1,
by having both the lowest mean error of 0.0021 and std error of 0.0002
```

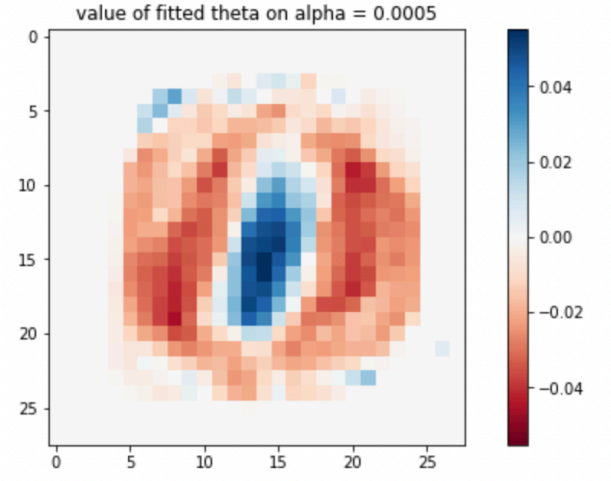
Q32.

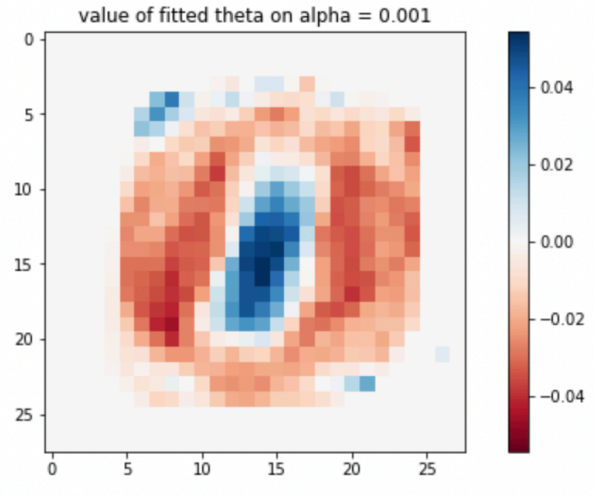
Finally, for one run of the fit for each value of α plot the value of the fitted θ. You can access it via clf.coef , and should reshape the 764 dimensional vector to a 28 × 28 arrray to visualize it with plt.imshow. Defining scale = np.abs(clf.coef).max(), you can use the following keyword arguments (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.

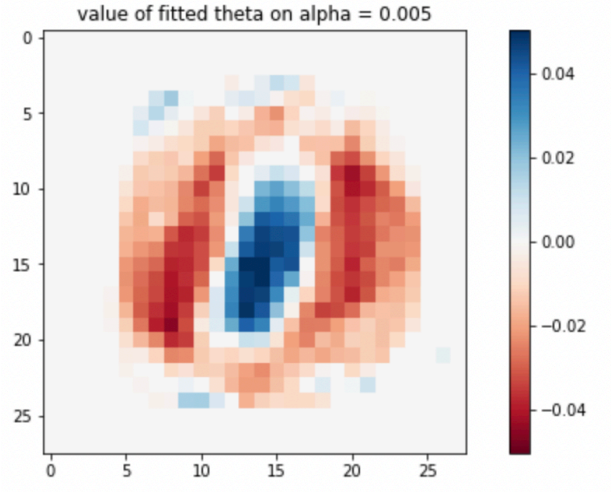
```
# Finally, for one run of the fit for each value of α plot the value of the fitted θ
alpha_list = [0.0001, 0.0005, 0.0007, 0.001, 0.005, 0.007, 0.01, 0.05, 0.07, 0.1]

# iterate by each alpha
for alpha in alpha_list:
    # create clf
    clf = SGDClassifier(loss='log', max_iter=1000,
                    tol=1e-3,
                    penalty='l1', alpha=alpha,
                    learning_rate='invscaling',
                    power_t=0.5,
                    eta0=0.01,
                    verbose=0)
    # fit
    clf.fit(X_train, y_train)

    # You can access it via clf.coef , and should reshape the 764 dimensional vector to a 28 × 28 arrray
    theta = clf.coef_
    theta_reshaped = theta.reshape(28,28)

    # 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.
    scale = np.abs(clf.coef_).max()

    # to visualize it with plt.imshow.

    plt.figure(figsize = (10,5))
    plt.imshow(theta_reshaped, cmap=plt.cm.RdBu, vmax=scale, vmin=-scale)
    plt.title('value of fitted theta on alpha = {}'.format(alpha))
    plt.colorbar()
    plt.show()
```

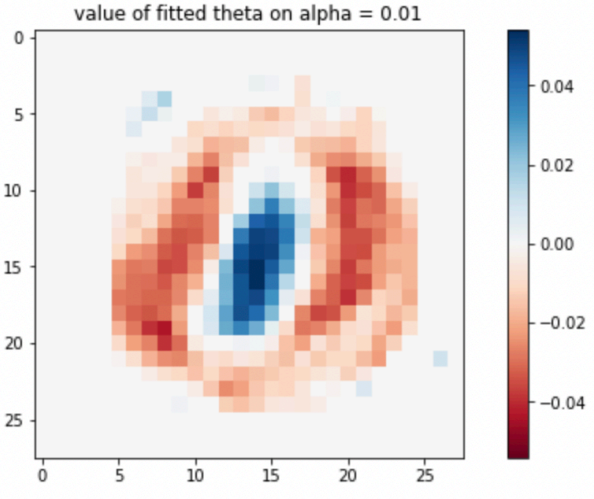

value of fitted theta on alpha = 0.0001

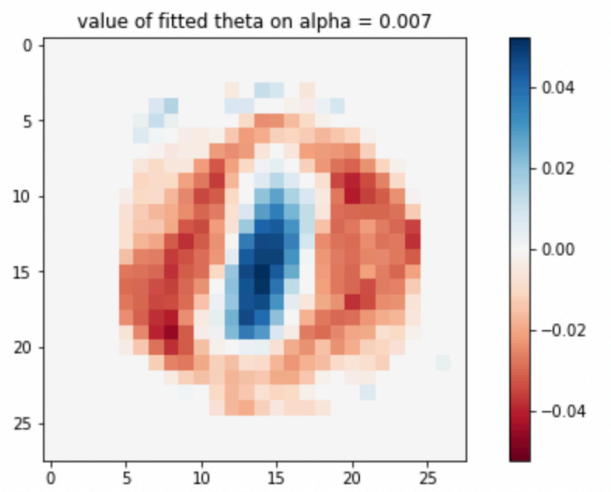

value of fitted theta on alpha = 0.0007

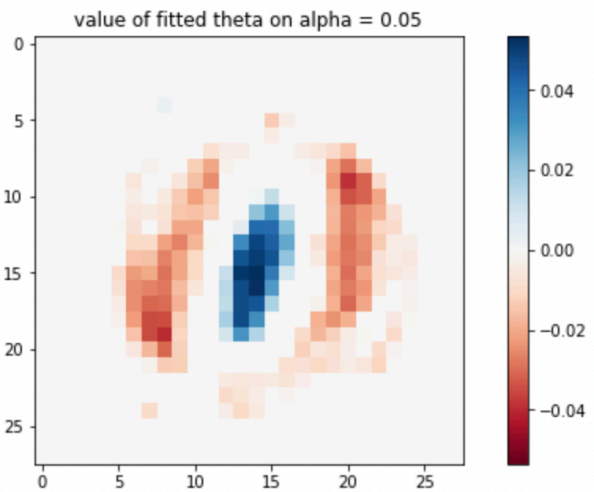

value of fitted theta on alpha = 0.0005

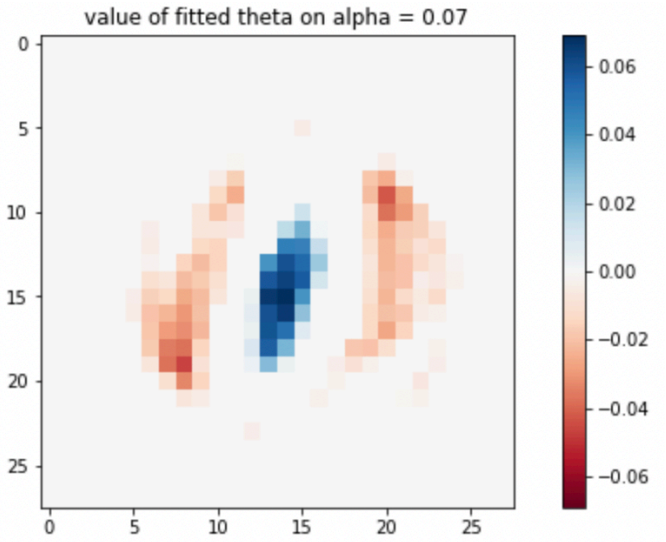

value of fitted theta on alpha = 0.001

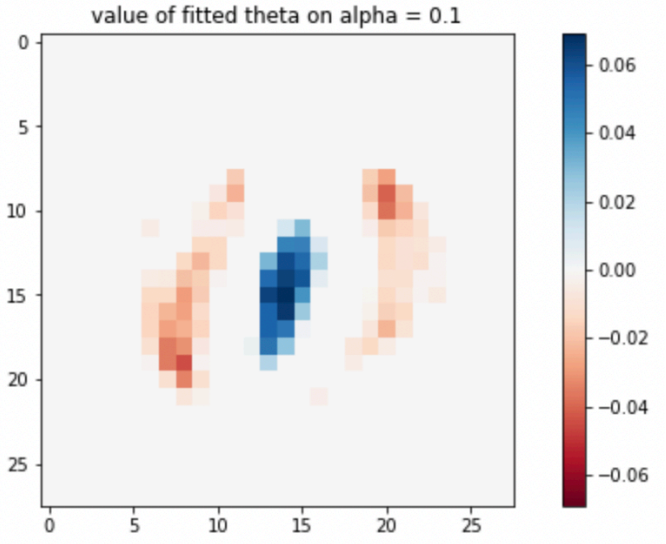

value of fitted theta on alpha = 0.005


value of fitted theta on alpha = 0.01


value of fitted theta on alpha = 0.007


value of fitted theta on alpha = 0.05





Q33.

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

Ans) As alpha gets larger, image '0' gets more clear and showing less blurred images.
This makes sense since we are using L1 regularization and L1 regularization acts as feature selection method. Larget regularization yields sparse coefficients, and L1 regularization makes coefficients shrinks towards to actual zeros. (So, there would be more zero coeeficients)