Homework 2: Gradient Descent & Regularization

Due: Wednesday, February 16, 2022 at 11:59PM

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better. The last application is optional.

This second homework features 3 problem sets and explores gradient descent algorithms, loss functions (both topics of week 2), regularization (topic of week 3) and statistical learning theory (week 1 + week 2). Following the instructions in the homework you should be able to solve a lot of questions before the lecture of week 3. Additionally this homework has an optional problem set. Optional questions will be graded but the points do not count towards this assignment. They instead contribute towards the extra credit you can earn over the entire course (maximum 2%) which can be used to improve the final grade by half a letter (e.g., A- to A).

1 Statistical Learning Theory

In the last HW, we used training error to determine whether our models have converged. It is crucial to understand what is the source of this training error. We specifically want to understand how it is connected to the noise in the data. In this question, we will compute the expected training error when we use least squares loss to fit a linear function.

Consider a full rank $N \times d$ data matrix X $(N > d)$ where the training labels are generated as $y_i = b\, x_i + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ is noise. From HW 1, we know the formula for the ERM, $\hat{b} = (X^T X)^{-1} X^T y$.

1. Show that:

$$\text{Training Error} = \frac{1}{N}\left|\left|\Big(X(X^TX)^{-1}X^T - I\Big)\epsilon\right|\right|_2^2$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2 I_n)$ and training error is defined as $\frac{1}{N}||X\hat{b} - y||_2^2$.

$$\begin{aligned}
\frac{1}{N}||X\hat{b} - y||_2^2 = & \\
&= \frac{1}{N}||X(X^TX)^{-1}X^Ty - y||_2^2 \\
&= \frac{1}{N}||X(X^TX)^{-1}X^T(Xb+\epsilon) - (Xb+\epsilon)||_2^2 \\
&= \frac{1}{N}||X(X^TX)^{-1}X^TXb + X(X^TX)^{-1}X^T\epsilon - Xb - \epsilon)||_2^2 \\
&= \frac{1}{N}||Xb + X(X^TX)^{-1}X^T\epsilon - Xb - \epsilon)||_2^2 \\
&= \frac{1}{N}||X(X^TX)^{-1}X^T\epsilon - \epsilon)||_2^2 \\
&= \frac{1}{N}||(X(X^TX)^{-1}X^T - I)\epsilon)||_2^2
\end{aligned}$$

2. Show that the expectation of the training error can be expressed solely in terms of **only** N, d, σ as:
$$E\Big[\frac{1}{N}\Big|\Big|\Big(X(X^TX)^{-1}X^T - I\Big)\epsilon\Big|\Big|_2^2\Big] = \frac{(N-d)}{N}\sigma^2$$

Hints:

- Consider $A = X(X^TX)^{-1}X^T$. What is A^TA? Is A symmetric? What is A^2?
- For a symmetric matrix A satisfying $A^2 = A$, what are its eigenvalues?
- If X is full rank, then what is the rank of A? What is the eigenmatrix of A?

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

$$\begin{aligned}
E[\frac{1}{N}\Big|\Big|\Big(A-I\Big)\epsilon\Big|\Big|_2^2] = &\\
&= \frac{1}{N}E\Big[\Big((A-I)\epsilon\Big)^T\Big((A-I)\epsilon\Big)\Big]\\
&= \frac{1}{N}E\Big[\Big(\epsilon^T(A-I)^T\Big)\Big((A-I)\epsilon\Big)\Big]\\
&= \frac{1}{N}E\Big[\Big(\epsilon^T(A^T-I^T)\Big)\Big((A-I)\epsilon\Big)\Big]\\
&= \frac{1}{N}E\Big[\Big(\epsilon^T(A-I)\Big)\Big((A-I)\epsilon\Big)\Big]\\
&= \frac{1}{N}E\Big[\Big(\epsilon^TA-\epsilon^T\Big)\Big((A\epsilon-\epsilon\Big)\Big]\\
&= \frac{1}{N}E\Big[\epsilon^TAA\epsilon-\epsilon^TA\epsilon-\epsilon^TA\epsilon+\epsilon^T\epsilon\Big]\\
&= \frac{1}{N}E\Big[\epsilon^TA\epsilon-2\epsilon^TA\epsilon+\epsilon^T\epsilon\Big]\\
&= \frac{1}{N}E\Big[\epsilon^T\epsilon-\epsilon^TA\epsilon\Big]\\
&= \frac{1}{N}\Big(E\Big[\sum_{i=1}^N\epsilon^2\Big]-E[\epsilon^TA\epsilon]\Big)\\
&= \frac{1}{N}\Big(\sum_{i=1}^N E[\epsilon^2]-(\mu_\epsilon^TA\mu_\epsilon+TR(A\Sigma\epsilon))]\Big)\\
&= \frac{1}{N}\Big(N\sigma^2-d\sigma^2\Big)\\
&= \frac{(N-d)}{N}\sigma^2
\end{aligned}$$

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

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

To clarify, the trace of $A\Sigma\epsilon$ consists of the eigenvalues of A multiplied by the variance of ϵ and the eigenvalues of A are 0 and 1 with rank d. Therefore there are $d\sigma^2$'s

3. From this result, give a reason as to why the training error is very low when d is close to N i.e. when we overfit the data.

   When d gets closer to N, the numerator approaches 0 and therefore the whole expression approaches 0. That is the reason why increasing the amount of features (even if meaningless) might end up reducing training error but testing error will increase and therefore show overfitting.

2 Gradient descent for ridge(less) linear regression

Dataset

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.

Feature normalization When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization, 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.

4. 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.

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`.

Linear regression

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.

5. 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.

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

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

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

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

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

$$\theta_i = \theta_i - \eta \times \frac{\partial J(\theta)}{\partial \theta}$$

8. 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.

9. 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.

Gradient checker

We can numerically check the gradient calculation. 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.

10. 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.

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

Batch gradient descent

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

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

12. 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.

13. 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.

Ridge Regression

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.

14. 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)

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

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

15. Implement `compute_regularized_square_loss_gradient.`

16. Implement `regularized_grad_descent.`

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, $\lambda \in \left\{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\right\}$. Then you can zoom in on the best range. Follow the steps below to proceed.

17. 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?

18. 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 ?

19. 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?

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

Stochastic Gradient Descent (SGD) (optional)

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.

21. 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.

In order to make the expressions equivalent, $f_i(\theta) = \left(h_\theta(\boldsymbol{x}_i) - y_i\right)^2 + \lambda\theta^T\theta$

22. 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 $\mathbb{E}\left[\nabla f_i(\theta)\right]$

$$\begin{aligned}
\nabla_\theta J_\lambda(\theta) &= E\big[\nabla f_i(\theta)\big] \\
&= E\big[\frac{2}{m}(X_i^T X_i\theta - X_i^T y_i) + 2\lambda\theta\big] \\
&= \big(2(X_i^T X_i\theta - X_i^T y_i) + 2\lambda\theta\big) \times P(X_i = x) \\
&= \big(2(X_i^T X_i\theta - X_i^T y_i) + 2\lambda\theta\big) \times \frac{1}{m} \\
&= \frac{2}{m}(X_i^T X_i\theta - X_i^T y_i) + 2\lambda\theta
\end{aligned}$$

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

$$\theta_i = \theta_i - \eta \times \frac{\partial J(\theta)}{\partial \theta}$$

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

24. Implement `stochastic_grad_descent`.

25. 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?

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.

Acknowledgement: This problem set is based on assignments developed by David Rosenberg of NYU and Bloomberg.

3 Image classification with regularized logistic regression

Dataset

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.

Logistic regression

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).

26. 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

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

The objective function for Logistic Regression, as seen in slide 45, is the following:

$$L(\theta) = \frac{1}{m} \sum_{i=1}^m log(1 + e^{-m_i}) = \frac{1}{m} \sum_{i=1}^m log(1 + e^{-y_i f_i(\theta)}) = \frac{1}{m} \sum_{i=1}^m log(1 + e^{-y_i(\theta^T x_i + b_i)})$$

Now, to prove it can be written as shown, we will plug in $y = 1$ and $y = -1$.

For $y = 1$:

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

For $y = -1$:

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

Now that we saw that after plugging in both values for y we obtain the original objective function, we can conclude that the it can therefore be written in that form.

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

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

$$L(\theta) = \frac{1}{m}\sum_{i=1}^{m} log(1 + e^{-y_i(\theta^T x_i + b_i)}) + \lambda \sum_{i=1}^{n} |\theta|$$

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.

28. 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)`.

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

29. 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.

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

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

32. 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.

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

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

In [1]:
```
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
```

Question 4

In [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
    """

    train = pd.DataFrame(train)
    test = pd.DataFrame(test)

    for feature in train.columns:
        if train[feature].nunique() == 1:
            train.drop(feature, axis=1)
            test.drop(feature, axis=1)

    min_val = np.min(train, axis=0)
    max_val = np.max(train, axis=0)

    train = (train - min_val) / (max_val - min_val)
    test = (test - min_val) / (max_val - min_val)

    return np.array(train), np.array(test)
```

In [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_sta

    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

X_train, y_train, X_test, y_test = load_data()
```

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

Question 8

In [4]:

```
def compute_square_loss(X, y, theta):
    """
    Given a set of X, y, theta, compute the average square loss for predicting y with X

    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.sum((X @ theta - y)**2)

    return loss
```

Question 9

In [5]:

```
def compute_square_loss_gradient(X, y, theta):
    """
    Compute the gradient of the average square loss(as defined in compute_square_loss),

    Args:
        X - the feature vector, 2D numpy array of size(num_instances, num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        theta - the parameter vector, 1D numpy array of size(num_features)

    Returns:
        grad - gradient vector, 1D numpy array of size(num_features)
    """

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

    return grad
```

Question 10

In [6]:

```
#Getting the gradient calculation correct is often the trickiest part of any gradient-b
#Fortunately, it's very easy to check that the gradient calculation is correct using th
#See http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimizati
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

    I = np.identity(theta.shape[0])

    for i in range(0, num_features):
        approx_grad[i] = (compute_square_loss(X, y, theta + epsilon * I[:,i]) - compute
    distance = np.sqrt(np.sum((true_gradient - approx_grad)**2))

    return distance
    if distance < tolerance:
        return True #There is no major difference between gradients
    else:
        return False #There is a major difference between gradients
```

In [7]:

```
def generic_gradient_checker(X, y, theta, objective_func, gradient_func,
                             epsilon=0.01, tolerance=1e-4):
    """
    The functions takes objective_func and gradient_func as parameters.
    And check whether gradient_func(X, y, theta) returned the true
    gradient for objective_func(X, y, theta).
    Eg: In LSR, the objective_func = compute_square_loss, and gradient_func = compute_s
    """
    #TODO
```

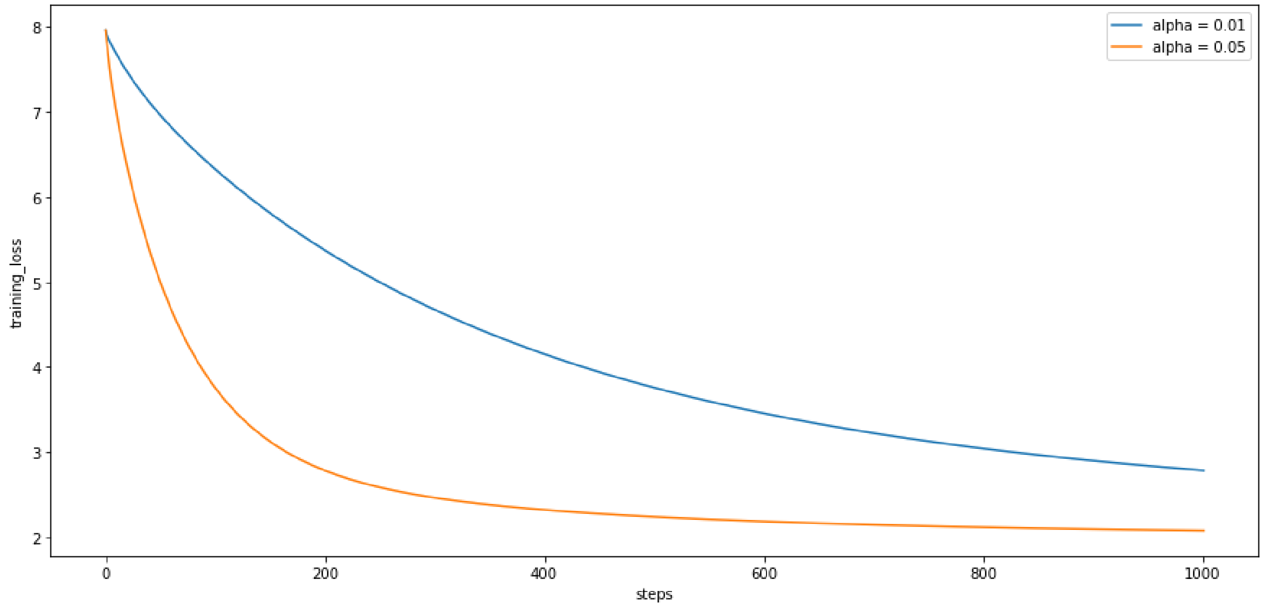
Question 11

In [8]:

```
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 upda

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array of size(num_step+1
                     for instance, theta in step 0 should be theta_hist[0], theta in st
        loss_hist - the history of average square loss on the data, 1D numpy array,(num
    """

    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

    loss_hist[0] = compute_square_loss(X, y, theta)
    theta_hist[0] = theta
    for i in range(1, num_step+1):

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

        loss = compute_square_loss(X, y, theta)
        loss_hist[i] = loss
        theta_hist[i] = theta

    return theta_hist, loss_hist
```

Question 12

In [9]:
```
x = np.linspace(0, 1001, 1001)

plt.figure(figsize=(15,7))

theta_hist, loss_hist = batch_grad_descent(X_train, y_train, alpha=0.01, num_step=1000,
sns.lineplot(x=x, y=loss_hist, label='alpha = 0.01')

theta_hist, loss_hist = batch_grad_descent(X_train, y_train, alpha=0.05, num_step=1000,
sns.lineplot(x=x, y=loss_hist, label='alpha = 0.05')

#theta_hist, loss_hist = batch_grad_descent(X_train, y_train, alpha=0.5, num_step=1000,
#sns.lineplot(x=x, y=theta_hist[:,0], label='alpha = 0.5')

#theta_hist, loss_hist = batch_grad_descent(X_train, y_train, alpha=0.1, num_step=1000,
#sns.lineplot(x=x, y=theta_hist[:,0], label='alpha = 0.5')

plt.xlabel('steps')
plt.ylabel('training_loss')
plt.show()
```

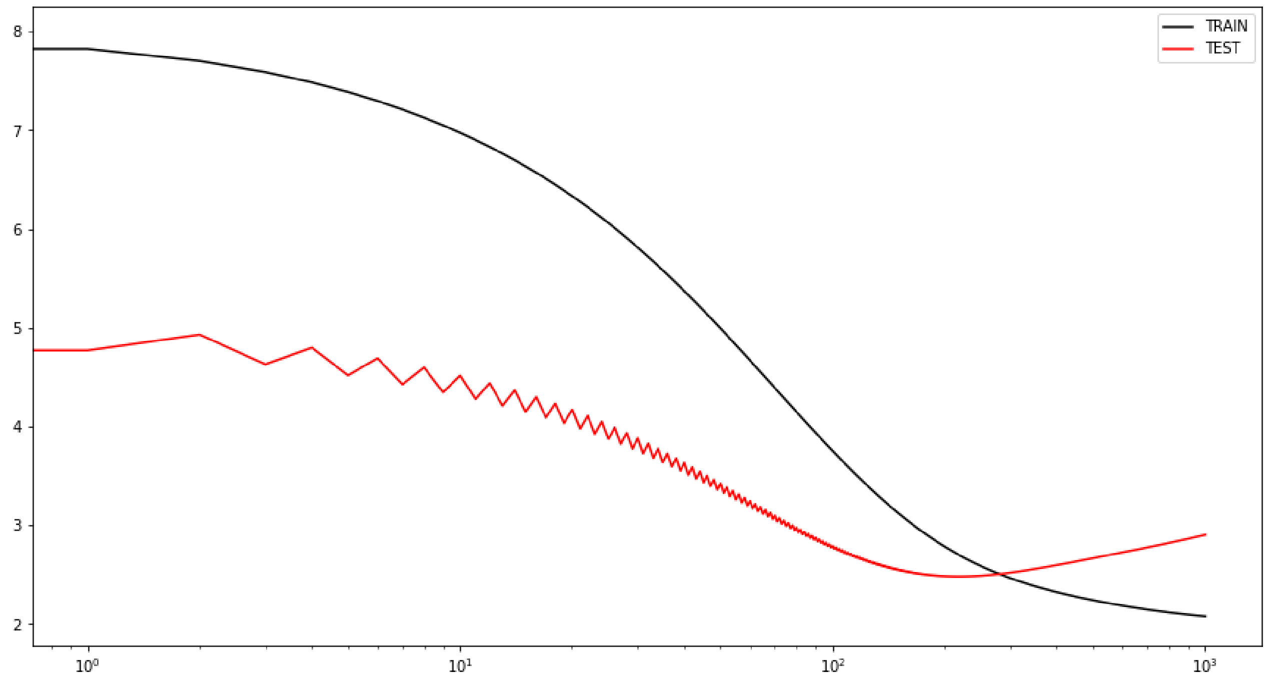


The step sizes 0.01 and 0.05 converge but 0.1 and 0.5 diverge. Additionally, we can see that the minimum loss is around 2.

Question 13

In [10]:

```
plt.figure(figsize=(15,8))

theta_hist, loss_hist = batch_grad_descent(X_train, y_train, alpha=0.05, num_step=1000,

test_loss = []
for i in range(len(theta_hist)):
    test_loss.append(compute_square_loss(X_test, y_test, theta_hist[i]))

plt.xscale('log')
plt.plot(x, loss_hist, label='TRAIN', color='black')
plt.plot(x, test_loss, label='TEST', color='red')
plt.legend()
plt.show()
```



Question 15

In [11]:

```python
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

    Args:
        X - the feature vector, 2D numpy array of size(num_instances, num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        theta - the parameter vector, 1D numpy array of size(num_features)
        lambda_reg - the regularization coefficient

    Returns:
        grad - gradient vector, 1D numpy array of size(num_features)
    """
    grad = (2 / X.shape[0]) * (X.T @ X @ theta - X.T @ y) + 2 * lambda_reg * theta

    return grad
```

Question 16

In [12]:

```python
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
                     for instance, theta in step 0 should be theta_hist[0], theta in st
        loss hist - the history of average square loss function without the regularizat
    """
    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

    loss_hist[0] = compute_square_loss(X, y, theta)
    theta_hist[0] = theta
    for i in range(1, num_step+1):

        theta = theta - alpha * compute_regularized_square_loss_gradient(X, y, theta, l
        loss = compute_square_loss(X, y, theta)
        loss_hist[i] = loss
        theta_hist[i] = theta

    return theta_hist, loss_hist
```

Question 17

In [13]:

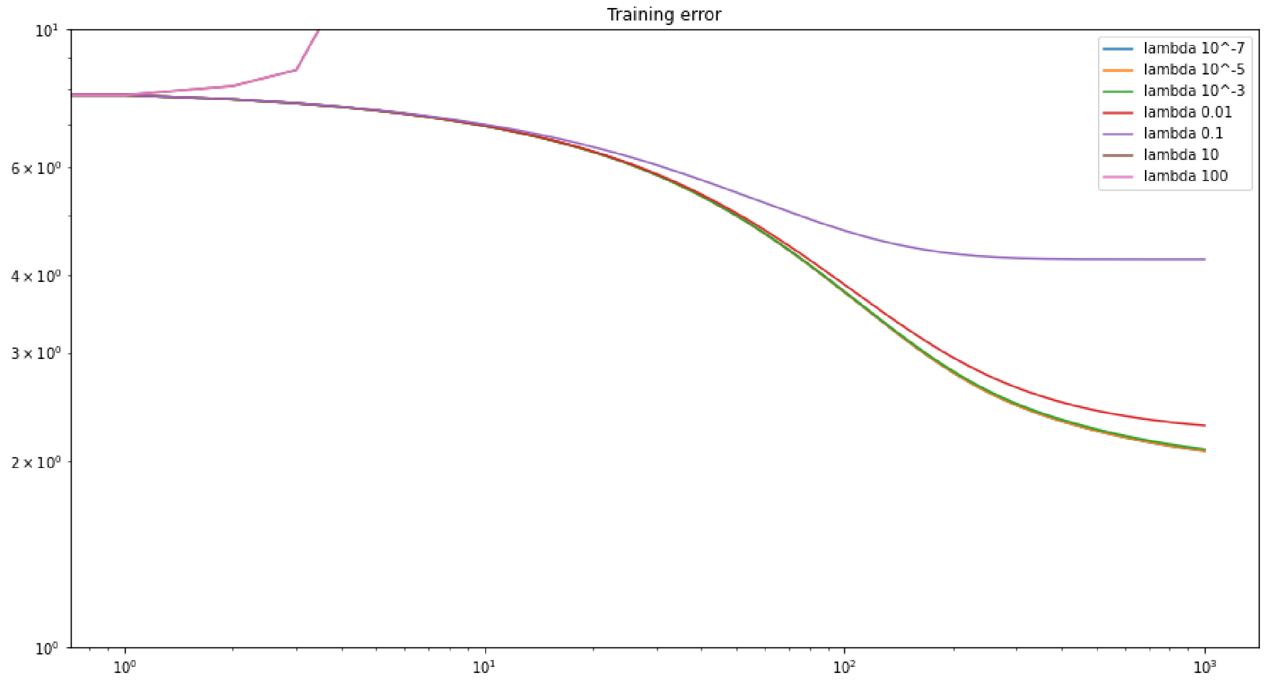
```
def plot_test_loss(X_train, y_train, X_test, y_test, lambda_reg):

    theta_hist, loss_hist = regularized_grad_descent(X_train, y_train, lambda_reg=lambd
    test_loss = []
    for i in range(len(theta_hist)):
        test_loss.append(compute_square_loss(X_test, y_test, theta_hist[i]))

    return test_loss
```

In [14]:

```
plt.figure(figsize=(15,8))

theta_hist, loss_hist_1 = regularized_grad_descent(X_train, y_train, alpha=0.05, lambda
theta_hist, loss_hist_2 = regularized_grad_descent(X_train, y_train, alpha=0.05, lambda
theta_hist, loss_hist_3 = regularized_grad_descent(X_train, y_train, alpha=0.05, lambda
theta_hist, loss_hist_4 = regularized_grad_descent(X_train, y_train, alpha=0.05, lambda
theta_hist, loss_hist_5 = regularized_grad_descent(X_train, y_train, alpha=0.05, lambda
theta_hist, loss_hist_6 = regularized_grad_descent(X_train, y_train, alpha=0.05, lambda
theta_hist, loss_hist_7 = regularized_grad_descent(X_train, y_train, alpha=0.05, lambda

sns.lineplot(x=x, y=loss_hist_1, label='lambda 10^-7')
sns.lineplot(x=x, y=loss_hist_2, label='lambda 10^-5')
sns.lineplot(x=x, y=loss_hist_3, label='lambda 10^-3')
sns.lineplot(x=x, y=loss_hist_4, label='lambda 0.01')
sns.lineplot(x=x, y=loss_hist_5, label='lambda 0.1')
sns.lineplot(x=x, y=loss_hist_6, label='lambda 10')
sns.lineplot(x=x, y=loss_hist_6, label='lambda 100')

plt.xscale('log')
plt.yscale('log')
plt.title('Training error')
plt.show()
```

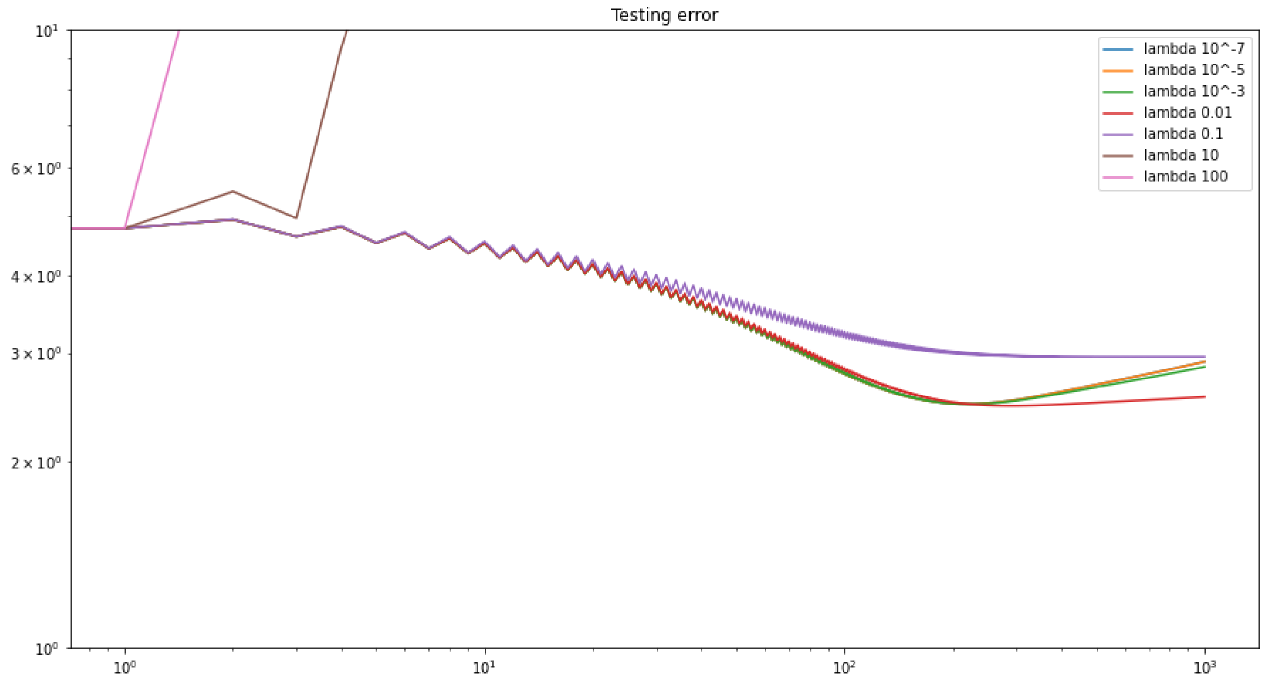
```
C:\Users\alexx\anaconda3\lib\site-packages\numpy\core\fromnumeric.py:87: RuntimeWarning:
overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/325647661.py:14: RuntimeWarning: overfl
ow encountered in square
  loss = (1 / X.shape[0]) * np.sum((X @ theta - y)**2)
```

```
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/2209270210.py:14: RuntimeWarning: overf
low encountered in matmul
  grad = (2 / X.shape[0]) * (X.T @ X @ theta - X.T @ y) + 2 * lambda_reg * theta
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/325647661.py:14: RuntimeWarning: invali
d value encountered in matmul
  loss = (1 / X.shape[0]) * np.sum((X @ theta - y)**2)
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/2209270210.py:14: RuntimeWarning: overf
low encountered in multiply
  grad = (2 / X.shape[0]) * (X.T @ X @ theta - X.T @ y) + 2 * lambda_reg * theta
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/2214137038.py:24: RuntimeWarning: inval
id value encountered in subtract
  theta = theta - alpha * compute_regularized_square_loss_gradient(X, y, theta, lambda_r
eg)
C:\Users\alexx\anaconda3\lib\site-packages\numpy\ma\core.py:6846: RuntimeWarning: overfl
ow encountered in power
  result = np.where(m, fa, umath.power(fa, fb)).view(basetype)
```



In [15]:
```python
plt.figure(figsize=(15,8))

test_loss_1 = plot_test_loss(X_train, y_train, X_test, y_test, lambda_reg=10**-7)
test_loss_2 = plot_test_loss(X_train, y_train, X_test, y_test, lambda_reg=10**-5)
test_loss_3 = plot_test_loss(X_train, y_train, X_test, y_test, lambda_reg=10**-3)
test_loss_4 = plot_test_loss(X_train, y_train, X_test, y_test, lambda_reg=0.01)
test_loss_5 = plot_test_loss(X_train, y_train, X_test, y_test, lambda_reg=0.1)
test_loss_6 = plot_test_loss(X_train, y_train, X_test, y_test, lambda_reg=10)
test_loss_7 = plot_test_loss(X_train, y_train, X_test, y_test, lambda_reg=100)

sns.lineplot(x=x, y=test_loss_1, label='lambda 10^-7')
sns.lineplot(x=x, y=test_loss_2, label='lambda 10^-5')
sns.lineplot(x=x, y=test_loss_3, label='lambda 10^-3')
sns.lineplot(x=x, y=test_loss_4, label='lambda 0.01')
sns.lineplot(x=x, y=test_loss_5, label='lambda 0.1')
sns.lineplot(x=x, y=test_loss_6, label='lambda 10')
sns.lineplot(x=x, y=test_loss_7, label='lambda 100')

plt.xscale('log')
plt.yscale('log'),
plt.title('Testing error')
plt.show()
```

```
C:\Users\alexx\anaconda3\lib\site-packages\numpy\core\fromnumeric.py:87: RuntimeWarning:
overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/325647661.py:14: RuntimeWarning: overfl
ow encountered in square
  loss = (1 / X.shape[0]) * np.sum((X @ theta - y)**2)
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/2209270210.py:14: RuntimeWarning: overf
low encountered in matmul
  grad = (2 / X.shape[0]) * (X.T @ X @ theta - X.T @ y) + 2 * lambda_reg * theta
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/325647661.py:14: RuntimeWarning: invali
d value encountered in matmul
  loss = (1 / X.shape[0]) * np.sum((X @ theta - y)**2)
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/2209270210.py:14: RuntimeWarning: overf
low encountered in multiply
  grad = (2 / X.shape[0]) * (X.T @ X @ theta - X.T @ y) + 2 * lambda_reg * theta
C:\Users\alexx\AppData\Local\Temp/ipykernel_5696/2214137038.py:24: RuntimeWarning: inval
id value encountered in subtract
  theta = theta - alpha * compute_regularized_square_loss_gradient(X, y, theta, lambda_r
eg)
C:\Users\alexx\anaconda3\lib\site-packages\numpy\ma\core.py:6846: RuntimeWarning: overfl
ow encountered in power
  result = np.where(m, fa, umath.power(fa, fb)).view(basetype)
```

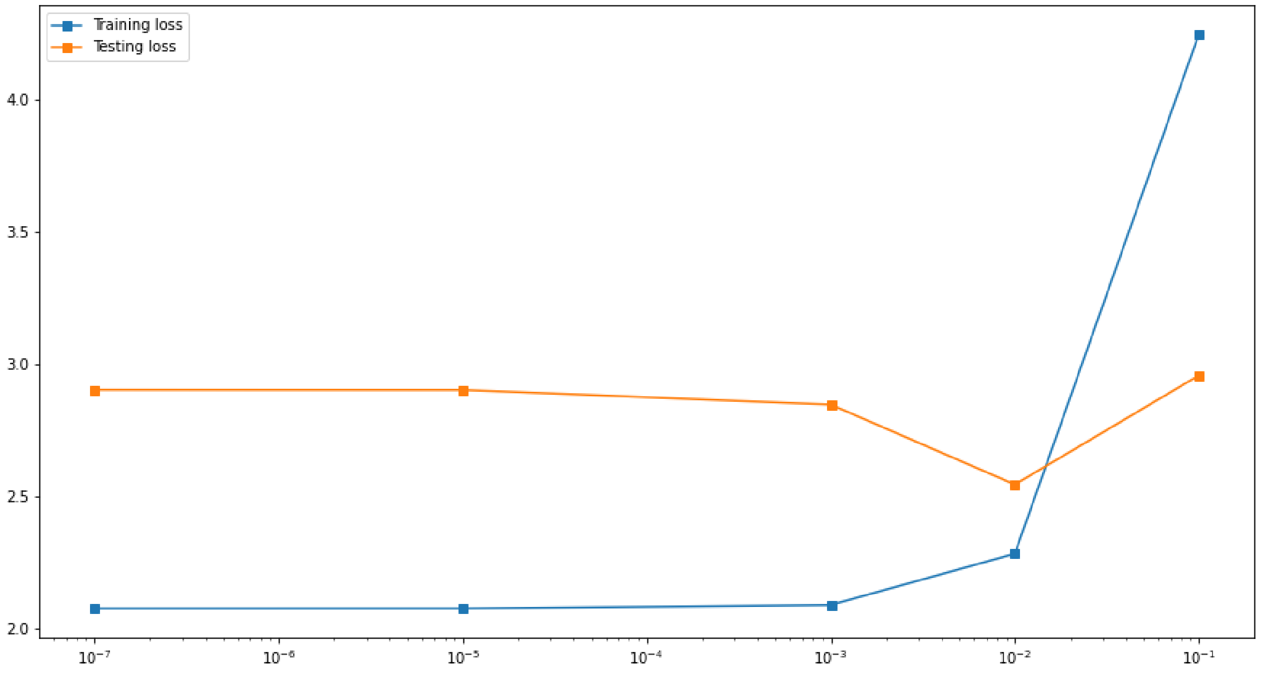


Overfitting consists on the training error being very low and the testing error being high, meaning that the training data was memorized and therefore the model fails to generalize. In this case, we can see that around step 400, the testing error starts increasing while the training error keeps going down, that when overfitting starts.

Question 18

In [16]:
```
plt.figure(figsize=(15,8))

a = [loss_hist_1[-1], loss_hist_2[-1], loss_hist_3[-1], loss_hist_4[-1], loss_hist_5[-1
```

```
b = [test_loss_1[-1], test_loss_2[-1], test_loss_3[-1], test_loss_4[-1], test_loss_5[-1
lambdas = [10**-7, 10**-5, 10**-3, 0.01, 0.1]

plt.plot(lambdas, a, label='Training loss', marker='s')
plt.plot(lambdas, b, label='Testing loss', marker='s')
plt.xscale('log')
plt.legend()
plt.show()
```

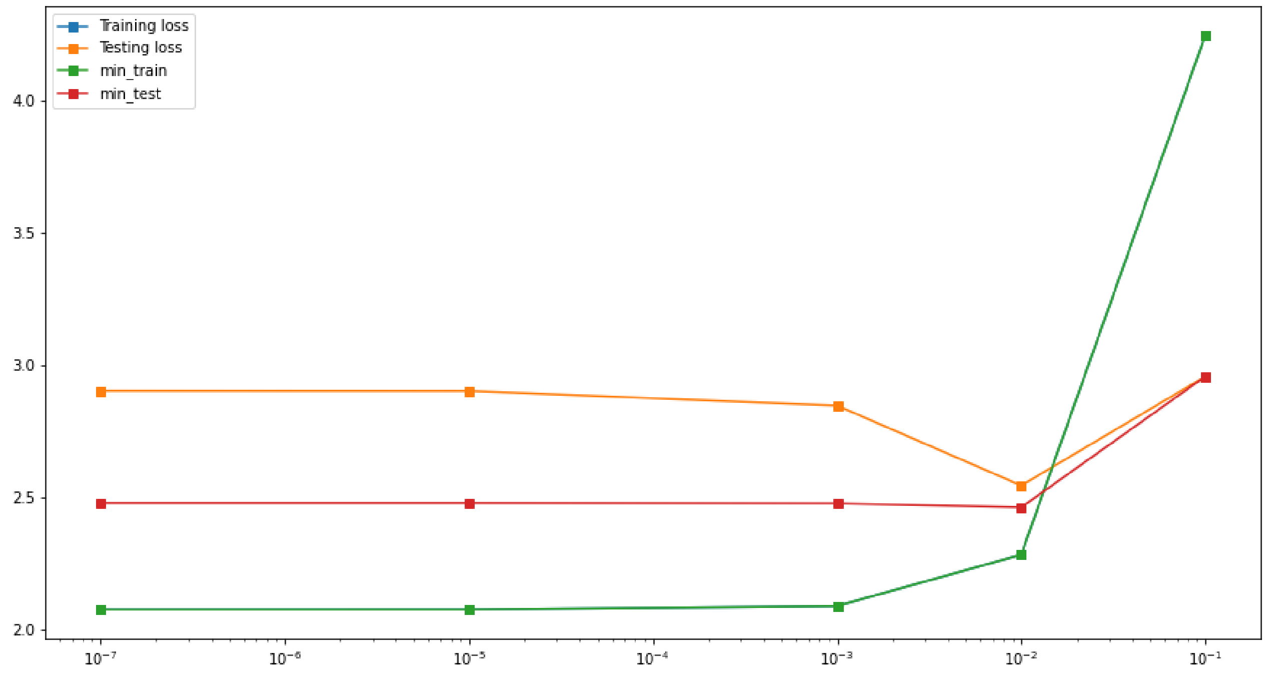


The best lambda would be where the training error intersects the testing error. Therefore, out of these lambdas I would choose lambda=0.01

Question 19

In [17]:

```
plt.figure(figsize=(15,8))

a = [loss_hist_1[-1], loss_hist_2[-1], loss_hist_3[-1], loss_hist_4[-1], loss_hist_5[-1
b = [test_loss_1[-1], test_loss_2[-1], test_loss_3[-1], test_loss_4[-1], test_loss_5[-1
min_a = [min(loss_hist_1), min(loss_hist_2), min(loss_hist_3), min(loss_hist_4), min(lo
min_b = [min(test_loss_1), min(test_loss_2), min(test_loss_3), min(test_loss_4), min(te
lambdas = [10**-7, 10**-5, 10**-3, 0.01, 0.1]

plt.plot(lambdas, a, label='Training loss', marker='s')
plt.plot(lambdas, b, label='Testing loss', marker='s')
plt.plot(lambdas, min_a, label='min_train', marker='s')
plt.plot(lambdas, min_b, label='min_test', marker='s')
plt.xscale('log')
plt.legend()
plt.show()
```



Now, the diference between testing errors is much lower so it does not matter as much but it does seem like the best lambda is still 0.01

Additionally, due to the fact that the training error always decreases, the minimum loss in training error is the same as the last training error (the green and blue line overlap).

Question 20

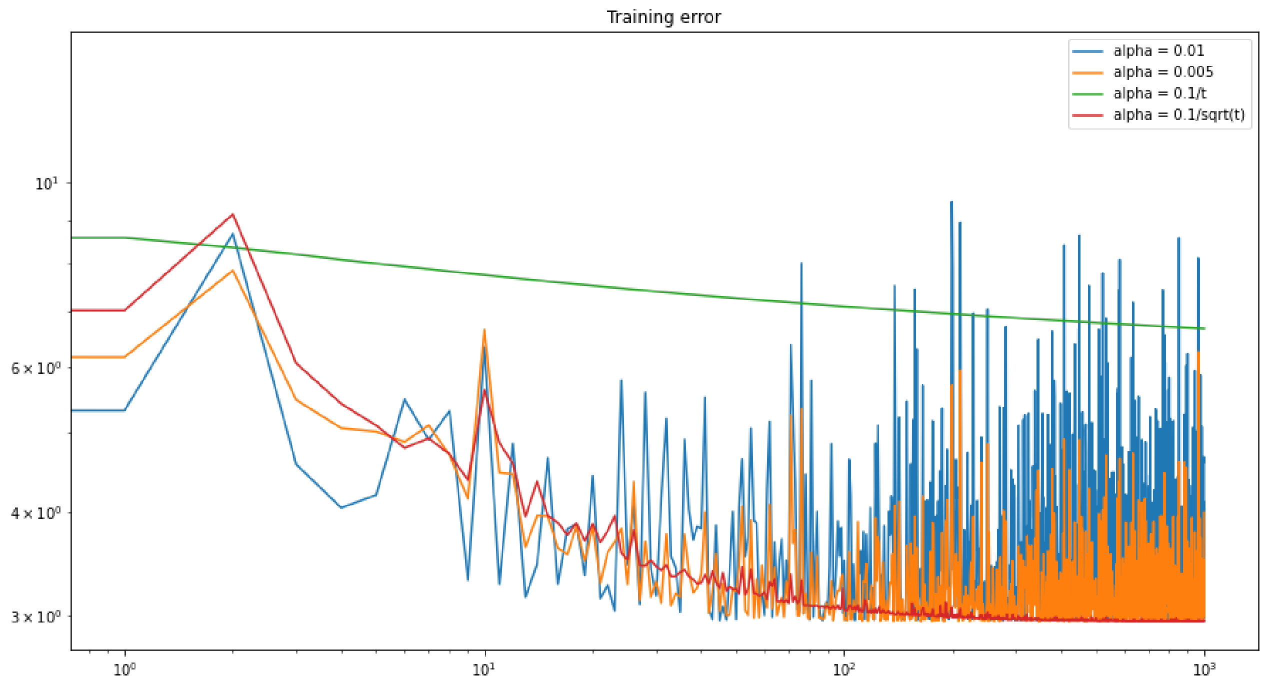
In practice, I would choose the theta that minimizes the test loss. In this case that is when alpha is 0.05, lambda is 0.01 and we use early stopping.

Question 24

In [18]:
```
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2, num_epoch=1000, eta0=F
    """
    In this question you will implement stochastic gradient descent with regularization

    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
                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,
                     for instance, theta in epoch 0 should be theta_hist[0], theta in e
        loss hist - the history of loss function vector, 2D numpy array of size(num_epo
    """
    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_h
    loss_hist = np.zeros((num_epoch, num_instances)) #Initialize loss_hist
    np.random.seed(27)

    for epoch in range(0, num_epoch):

        #random_index = np.random.choice(list(range(0, len(X))), len(X), replace=False)
        random_index = np.random.permutation(num_instances)
        for i in range(0, len(random_index)):

            #Save the historical loss and thetas
            loss_hist[epoch, i] = compute_square_loss(X, y, theta) + lambda_reg * theta
            theta_hist[epoch, i] = theta

            X_temp = X[random_index[i]].reshape(1,49)
            y_temp = y[random_index[i]].reshape(1,1)
            theta = theta.reshape(49,1)

            grad = 2 * (X_temp.T @ X_temp @ theta - (X_temp.T @ y_temp)) + 2 * lambda_r

            grad = grad.reshape(49,)
            theta = theta.reshape(49,)

            t = epoch * 100 + i+1
            #Check alpha
            if alpha == "1/sqrt(t)":
                theta = theta - 0.1/np.sqrt(t) * grad
            elif alpha == "1/t":
                theta = theta - 0.1/t * grad
            else:
                theta = theta - alpha * grad

    return theta_hist, loss_hist
```

In [19]:

```
plt.figure(figsize=(15,8))
x = np.linspace(0, 1000, 1000)

theta_hist1, loss_hist1 = stochastic_grad_descent(X_train, y_train, alpha=0.01, lambda_
theta_hist2, loss_hist2 = stochastic_grad_descent(X_train, y_train, alpha=0.005, lambda
theta_hist3, loss_hist3 = stochastic_grad_descent(X_train, y_train, alpha="1/t", lambda
theta_hist4, loss_hist4 = stochastic_grad_descent(X_train, y_train, alpha="1/sqrt(t)",
theta_hist5, loss_hist5 = stochastic_grad_descent(X_train, y_train, alpha="1/sqrt(t)",

sns.lineplot(x=x, y=loss_hist1[:,-1], label='alpha = 0.01')
sns.lineplot(x=x, y=loss_hist2[:,-1], label='alpha = 0.005')
sns.lineplot(x=x, y=loss_hist3[:,-1], label='alpha = 0.1/t')
sns.lineplot(x=x, y=loss_hist4[:,-1], label='alpha = 0.1/sqrt(t)')

plt.xscale('log')
plt.yscale('log')
plt.title('Training error')
plt.show()
```



Question 25

It is clear that when the step size is constant there is much moire noise. Additionally, using a step size of C/sqrt(t) appears to be the best alpha.

Question 28

In [20]:
```
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
```

In [21]:
```
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
```

In [22]:

```
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]
```

In [23]:

```
def classification_error(clf, X, y):
    return (np.sum(abs(clf.predict(X) - y))) / (2*X.shape[0])
```

In [24]:

```
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=1)
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)
```

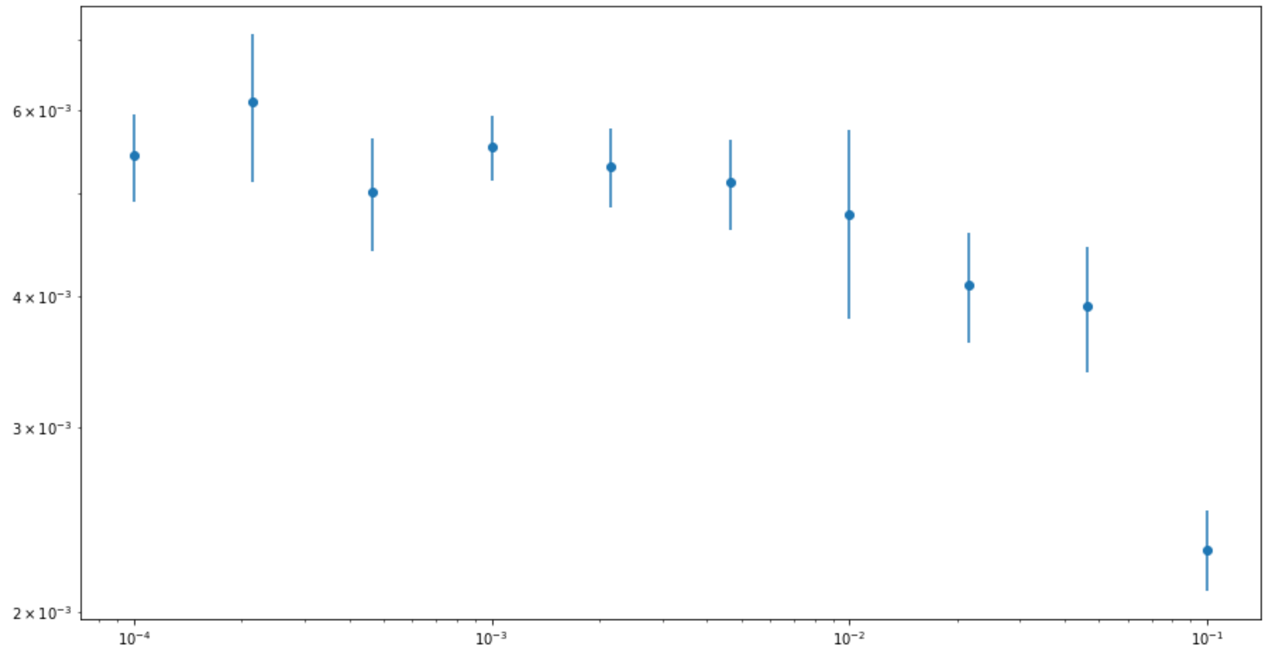
```
-- Epoch 1
Norm: 0.68, NNZs: 264, Bias: 0.008605, T: 9902, Avg. loss: 0.042496
Total training time: 0.03 seconds.
-- Epoch 2
Norm: 0.77, NNZs: 245, Bias: 0.008298, T: 19804, Avg. loss: 0.031968
Total training time: 0.06 seconds.
-- Epoch 3
Norm: 0.83, NNZs: 234, Bias: 0.008331, T: 29706, Avg. loss: 0.030265
Total training time: 0.10 seconds.
-- Epoch 4
Norm: 0.88, NNZs: 225, Bias: 0.008497, T: 39608, Avg. loss: 0.029343
Total training time: 0.13 seconds.
-- Epoch 5
Norm: 0.92, NNZs: 221, Bias: 0.008752, T: 49510, Avg. loss: 0.028770
Total training time: 0.17 seconds.
-- Epoch 6
Norm: 0.95, NNZs: 217, Bias: 0.009043, T: 59412, Avg. loss: 0.028308
```

```
Total training time: 0.20 seconds.
-- Epoch 7
Norm: 0.98, NNZs: 214, Bias: 0.009365, T: 69314, Avg. loss: 0.028039
Total training time: 0.23 seconds.
-- Epoch 8
Norm: 1.01, NNZs: 212, Bias: 0.009708, T: 79216, Avg. loss: 0.027815
Total training time: 0.27 seconds.
Convergence after 8 epochs took 0.27 seconds
train:  0.0021207836800646333   test:  0.001025010250102501
```

Question 29

In [25]:
```
mean_list, std_list = [], []
alpha_list = np.logspace(-4,-1, 10)


for alpha in alpha_list:

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

    error_list = []
    for _ in range(0, 10):
        X_train, y_train = sub_sample(100, X_train, y_train)
        clf.fit(X_train, y_train)

        error_list.append(classification_error(clf, X_test, y_test))

    mean_list.append(np.mean(error_list))
    std_list.append(np.std(error_list))
```

In [26]:
```
plt.figure(figsize = (15,8))

plt.errorbar(alpha_list,mean_list,yerr = std_list, fmt = 'o')

plt.yscale("log")
plt.xscale('log')
plt.show()
```



Question 30

The sole source of randomness comes from the selection of the subset of data.

Question 31

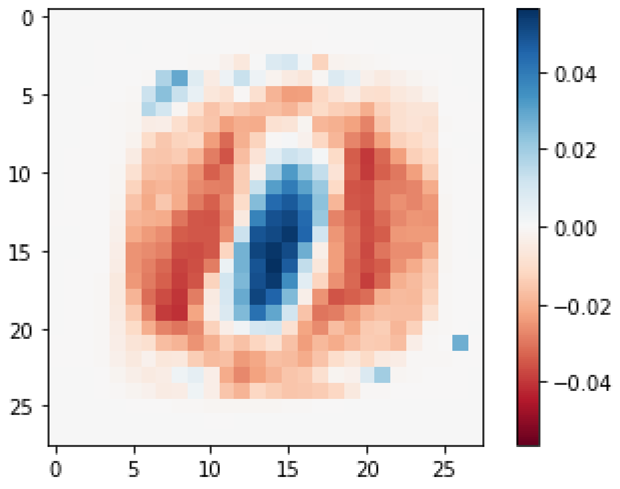
The optimal value of the parameter alpha is 0.1, it has the lowest test error mean as seen in the graph.

Question 32

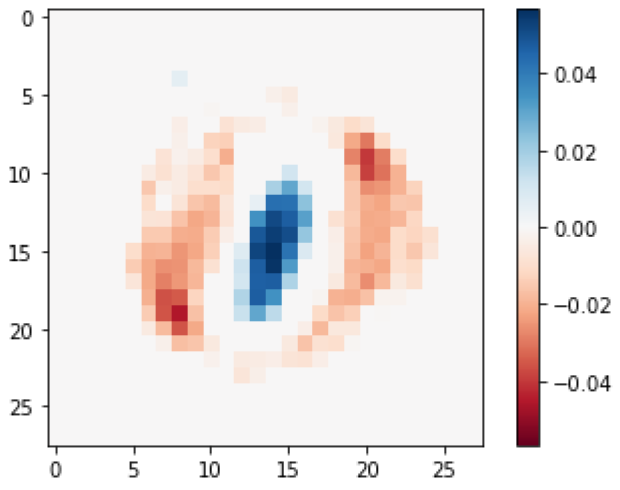
In [27]:
```
clf = SGDClassifier(loss='log', max_iter=1000, tol=1e-3, alpha=0,
                        learning_rate='invscaling', power_t=0.5, eta0=0.01)
clf.fit(X_train, y_train)
coeffs = clf.coef_.reshape(28,28)
```

In [28]:
```
clf_reg = SGDClassifier(loss='log', max_iter=1000, tol=1e-3, penalty='l1', alpha=0.05,
                        learning_rate='invscaling', power_t=0.5, eta0=0.01)
clf_reg.fit(X_train, y_train)
coeffs_reg = clf_reg.coef_.reshape(28,28)
```

In [29]:
```
scale = np.abs(clf.coef_).max()
plt.imshow(coeffs, cmap=plt.cm.RdBu, vmax=scale, vmin=-scale)
plt.colorbar()
plt.show()
```



In [30]:
```
scale = np.abs(clf_reg.coef_).max()
plt.imshow(coeffs_reg, cmap=plt.cm.RdBu, vmax=scale, vmin=-scale)
plt.colorbar()
plt.show()
```



The pattern of coefficients reflect the pixels of the image, we can see that the value is lower (red) when the pixel is assigned to the target 0 and higher (blue) when it is assigned to the target 1.

Additionallty, the regularization turns the less informative coefficients to 0 and therefore they are not seen in the second graph.