DS-GA 1003 HW2

Buz Galbraith

TOTAL POINTS

33.25 / 41

QUESTION 1

1 Q1 2 / 2

✓ **- 0 pts** *Correct*

- 1 pts Incomplete Proof

- 2 pts Missing Proof

QUESTION 2

2 Q2 4 / 4

✓ **- 0 pts** *Correct*

- 2 pts incomplete proof

- 4 pts missing proof

QUESTION 3

3 Q3 1 / 1

✓ **- 0 pts** *Correct*

- 1 pts incorrect

QUESTION 4

4 Q4 0 / 1

- 0 pts Correct

✓ **- 1 pts** *incorrect implementation, pay attention to whether you used train max&min to normalize and whether the result normalizes to [0, 1]*

- 0.5 pts need to discard features that are constant in the training set before normalization

QUESTION 5

5 Q5 1 / 1

✓ **- 0 pts** *Correct*

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

QUESTION 6

6 Q6 1 / 1

✓ **- 0 pts** *Correct*

- 0 pts Consistent with previous expression

- 0.5 pts $$\nabla J = \frac{2}{m}X^\top (X\theta - y)$$

- 0.5 pts Transposed result; the gradient of $$J$$ is a column vector

QUESTION 7

7 Q7 1 / 1

✓ **- 0 pts** *Correct*

- 1 pts Update is done on $$\theta$$

- 1 pts Update needs to go at the opposite direction of the gradient

QUESTION 8

8 Q8 1 / 1

✓ **- 0 pts** *Correct*

- 0 pts Consistent with previous result

- 1 pts Missing submission

QUESTION 9

9 Q9 1 / 1

✓ **- 0 pts** *Correct*

- **0 pts** Consistent
- **0.5 pts** Factor 2 missing

QUESTION 10

10 Q10 **1.5 / 2**

- **0 pts** Correct
- **0.5 pts** Should use the norm or element wise
- **0.5 pts** If using the norm, should take the norm of the difference

✓ **- 0.5 pts** *Missing a square root for the distance*

QUESTION 11

11 Q11 **0.5 / 1**

- **0 pts** Correct

✓ **- 0.5 pts** *Partially functioning code*

- **1 pts** Code not functioning
- **0.25 pts** Other minor issue

💬 missing grad check

QUESTION 12

12 Q12 **0.75 / 1**

- **0 pts** Correct
- **0.5 pts** No plot or wrong plot
- **0.5 pts** No or wrong findings

✓ **- 0.25 pts** *Click here to replace this description.*

- **0.5 pts** Click here to replace this description.

💬 missing 0.5 case required by the question.

QUESTION 13

13 Q13 **1 / 1**

✓ **- 0 pts** *please plot for the only one learning rate you selected from Q12*

- **0 pts** Correct
- **1 pts** wrong plot. you should observe loss first decrease and then increase. the plot should also be smooth
- **0.5 pts** Click here to replace this description.

QUESTION 14

14 Q14 **0.5 / 1**

- **0 pts** Correct
- **0.5 pts** Wrong or missing gradient

✓ **- 0.5 pts** *Wrong or missing theta update expression*

💬 you don't need to take gradient again in your last expression

QUESTION 15

15 Q15 **1 / 1**

✓ **- 0 pts** *Correct*

- **0.5 pts** Partially functioning
- **1 pts** Not functioning
- **0 pts** Inefficient solution. Do matrix-vector operation instead. remember X.T is transpose of X

QUESTION 16

16 Q16 **1 / 2**

- **0 pts** Correct
- **0 pts** you can ignore grad_check for this question
- **0 pts** you can use previous functions directly

✓ **- 1 pts** *Partially functioning*

- **2 pts** Not functioning

💬 missing initial loss

QUESTION 17

17 Q17 **1 / 1**

✓ **- 0 pts** *Correct*

- **0.5 pts** Wrong plots
- **0.5 pts** Missing discussion
- **1 pts** empty/not readable

QUESTION 18

18 Q18 **2 / 2**

✓ **- 0 pts** *Correct*

- **1 pts** Missing plots
- **1 pts** Missing best lambda
- **0.5 pts** The best lambda should be around 0.02
- **1 pts** should select lambda based on testing loss
- **2 pts** empty/not readable

QUESTION 19

19 Q19 **1 / 1**

✓ **- 0 pts** *Correct*

- **0.5 pts** Missing early stopping curve
- **0.5 pts** Missing lambda
- **1 pts** empty/not readable

QUESTION 20

20 Q20 **0.5 / 1**

- **0 pts** Correct

✓ **- 0.5 pts** *Incomplete*

- **0.5 pts** Missing reason
- **0.5 pts** should not select theta based on training error
- **1 pts** should select theta that minimize test error
- **1 pts** empty/not readable

should specify it is the testing loss that we want to minimize

QUESTION 21

21 Q21 (Optional) **1 / 1**

✓ **- 0 pts** *Correct*

- **0.5 pts** partially wrong fi
- **1 pts** empty

QUESTION 22

22 Q22 (Optional) **1 / 1**

✓ **- 0 pts** *Correct*

- **0.5 pts** Partially correct
- **1 pts** No answer

QUESTION 23

23 Q23 (Optional) **0.5 / 1**

- **0 pts** Correct

✓ **- 0.5 pts** *$$\nabla f_i(\theta)$$ not calculated / miscalculated*

- **0.5 pts** Update function incorrect
- **1 pts** No answer

QUESTION 24

24 Q24 (Optional) **1 / 1**

✓ **- 0 pts** *Correct*

- **0.5 pts** Partially correct
- **1 pts** No Answer

QUESTION 25

25 Q25 (Optional) **1 / 1**

✓ **- 0 pts** *Correct*

- **0.5 pts** Missing plots
- **0.5 pts** Missing comparison and reasoning
- **0.5 pts** Flawed plots / reasoning
- **1 pts** No / wrong answer

QUESTION 26

26 Q26 **1 / 1**

✓ **- 0 pts** *Correct*

- 0.5 pts incomplete proof, need to show how y takes -1 and 1 leads to the final equation

- 1 pts missing proof

QUESTION 27

27 Q27 **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts incorrect, should be adding $$\alpha||\theta||_1$$ to the previous loss function

QUESTION 28

28 Q28 **0 / 1**

- 0 pts Correct

- 0.5 pts Partially correct

✓ **- 1 pts** *Wrong / no answer*

QUESTION 29

29 Q29 **2 / 2**

✓ **- 0 pts** *Correct.*

- 0.5 pts x-axis not in log scale.

- 0.5 pts Fewer than 10 alpha values in the specified range.

- 0.5 pts Missing some error bars.

- 1 pts Should use L1 regularization, not L2.

- 1 pts No error bars or wrong standard errors

- 1.5 pts No or wrong classification errors.

- 2 pts No answer shown.

QUESTION 30

30 Q30 **0.5 / 1**

- 0 pts Correct

✓ **- 0.5 pts** *The answer is too vague or partial. Sources of randomness should come from 1). the **random initialization** of the parameters θ and b*

*2). the **random shuffling** of the same 100 samples in SGD in each experiment*

*Both will cause the **gradient path** to differ every round.*

(Note we do not subsample data each time. Need to be clear about why SGDClassifier selects different paths each time by mentioning random shuffling/order.)

- 1 pts Wrong.

- 1 pts No answer shown.

QUESTION 31

31 Q31 **0.5 / 1**

- 0 pts Correct alpha with the minimum classification error.

✓ **- 0.5 pts** *The optimal value of alpha does not match the one with the lowest error in the plot, or no plot as reference.*

- 0.5 pts Should choose only one optimal value.

- 0.5 pts The trend of the plot itself is not correct.

- 1 pts No answer was shown.

QUESTION 32

32 Q32 **0.5 / 1**

- 0 pts Correct

✓ **- 0.5 pts** *Didn't display the 10 plots corresponding to 10 different values of alpha.*

- 0.5 pts Wrong plot type. Should use

plt.imshow() to show the plot in 2-d dimension.

- 0.5 pts Theta trend is not correct or not clear. Should be more sparse as alpha increases.

- 0.5 pts Wrong coef values or scales.

- Coef should be reshaped to 28x28 array.
- Use one appropriate scale for all plots to show trends better.

- 1 pts No answer shown.

QUESTION 33

33 Q33 **0.5 / 1**

- 0 pts Correct

✓ **- 0.5 pts** *Didn't mention or wrong about the pattern recognized by θ. (detecting the shape of positive and negative values from the image). Only talked about the effect of regularization.*

- 0.5 pts Didn't mention or wrong about the effect of regularization. (making the coefficients more sparse.)

- 1 pts Didn't mention the pattern of θ nor the effect of regularization on sparsity.

- 1 pts No answer shown.

Homework 2: Gradient Descent & Regularization

Due: Wednesday, February 15, 2023 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.

This second homework features 3 problems and explores statistical learning theory (week 1 + week 2), gradient descent algorithms, loss functions (both topics of week 2), and regularization (topic of week 3). Following the instructions in the homework you should be able to solve a lot of questions before the lecture of week 3. Additionally, some parts of this homework are optional. 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:
$$\text{Training Error} = \frac{1}{N} \Big|\Big| \Big(X(X^T X)^{-1} X^T - I \Big) \epsilon \Big|\Big|_2^2$$
where $\epsilon \sim \mathcal{N}(0, \sigma^2 I_n)$ and training error is defined as $\frac{1}{N}||X\hat{b} - y||_2^2$.

   - given that we can express training error as training error= $\frac{1}{N}||X\hat{b} - y||_2^2$ we can write
   - training error= $\frac{1}{N}||X\hat{b} - y||_2^2$
     $= \frac{1}{N}||X(X^T X)^{-1} X^T y - y||_2^2$
     $= \frac{1}{N}||(X(X^T X)^{-1} X^T - I) y||_2^2$
     $= \frac{1}{N}||(X(X^T X)^{-1} X^T - I)(Xb + \epsilon)||_2^2$
     $= \frac{1}{N}||X(X^T X)^{-1} X^T (Xb) - Xb + (X(X^T X)^{-1} X^T - I)(\epsilon)||_2^2$
     $= \frac{1}{N}||X(X^T X)^{-1} (X^T (X) b - Xb + (X(X^T X)^{-1} X^T - I)(\epsilon)||_2^2$
     $= \frac{1}{N}||Xb - Xb + (X(X^T X)^{-1} X^T - I)(\epsilon)||_2^2$
     $= \frac{1}{N}||(X(X^T X)^{-1} X^T - I)(\epsilon)||_2^2$ which was the desired result

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

1 Q1 **2 / 2**

✓ **- 0 pts** *Correct*

- 1 pts Incomplete Proof

- 2 pts Missing Proof

Homework 2: Gradient Descent & Regularization

Due: Wednesday, February 15, 2023 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.

This second homework features 3 problems and explores statistical learning theory (week 1 + week 2), gradient descent algorithms, loss functions (both topics of week 2), and regularization (topic of week 3). Following the instructions in the homework you should be able to solve a lot of questions before the lecture of week 3. Additionally, some parts of this homework are optional. 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:
$$\text{Training Error} = \frac{1}{N} \Big|\Big| \Big(X(X^T X)^{-1} X^T - I \Big) \epsilon \Big|\Big|_2^2$$
where $\epsilon \sim \mathcal{N}(0, \sigma^2 I_n)$ and training error is defined as $\frac{1}{N}||X\hat{b} - y||_2^2$.

   - given that we can express training error as training error= $\frac{1}{N}||X\hat{b} - y||_2^2$ we can write
   - training error= $\frac{1}{N}||X\hat{b} - y||_2^2$
     $= \frac{1}{N}||X(X^T X)^{-1} X^T y - y||_2^2$
     $= \frac{1}{N}||(X(X^T X)^{-1} X^T - I) y||_2^2$
     $= \frac{1}{N}||(X(X^T X)^{-1} X^T - I)(Xb + \epsilon)||_2^2$
     $= \frac{1}{N}||X(X^T X)^{-1} X^T (Xb) - Xb + (X(X^T X)^{-1} X^T - I)(\epsilon)||_2^2$
     $= \frac{1}{N}||X(X^T X)^{-1} (X^T (X) b - Xb + (X(X^T X)^{-1} X^T - I)(\epsilon)||_2^2$
     $= \frac{1}{N}||Xb - Xb + (X(X^T X)^{-1} X^T - I)(\epsilon)||_2^2$
     $= \frac{1}{N}||(X(X^T X)^{-1} X^T - I)(\epsilon)||_2^2$ which was the desired result

2. Show that the expectation of the training error can be expressed solely in terms of **only** N, d, σ as:
$$E\Big[\frac{1}{N} \Big|\Big| \Big(X(X^T X)^{-1} X^T - I \Big) \epsilon \Big|\Big|_2^2\Big] = \frac{(N-d)}{N} \sigma^2$$
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 us define $A = X(X^TX)^{-1}X^T$
  - first we can show that A is symmetric ie $A^t = A$
    - $A^t = (X(X^TX)^{-1}X^T)^T = (X^T)^T((X^TX)^{-1})^TX^T = (X^T)^T((X^TX)^T)^{-1}X^T = (X)((XX^T))^{-1}X^T = A$
  - next we can show that $A = A^TA = A^2$
    - as we showed above A is symmetric we know that $A^2 = A^TA$
    - thus we have $A^2 = A^TA = (X(X^TX)^{-1}X^T)(X(X^TX)^{-1}X^T) = X(X^TX)^{-1}(X^TX)(X^TX)^{-1}X^T = XI(X^TX)^{-1}X^T = X(X^TX)^{-1}X^T = A$
  - now we can reason about the eigenvalues of A
    - suppose $v \in (X)$ this would imply that $\exists u \in \mathbb{R}^{\mathbb{D}} Xu = v$ so consider $Av = AXu = X(X^TX)^{-1}X^TXu = X(X^TX)^{-1}(X^TX)u = X(X^TX)^{-1}X^TXu = Xu = v$ thus any vector $v \in img(X)$ is an eigenvector of A with eigenvalue 1.
    - we know that X is full rank meaning $dim(img(X)) = \mathbb{R}^d$ thus by rank nullity theorem $dim(ker(x)) = dim(ker(x^t)) = \mathbb{R}^{d-n}$
    - we can also see that $A \in \mathbb{R}^{N\times N}$ thus it is clear that A will have d eigenvectors with value 1.
    - we know on the other hand that if $v \in ker(x^t)$ we can write that $X^tv = 0$ thus $Av = X(X^TX)^{-1}X^Tv = 0$ meiang that $v \in ker(A)$ thus we know that A has n-d eigen values of value 0
    - this further implies that the $rank(X) = Rank(A) = d$
    - and finally that the Eigenmatrix of A can be expressed as $\begin{pmatrix} 1 & 0 & ... & 0 \\ 0 & 1.. & 0 & \\ 0 & 0 & ... & 0 \end{pmatrix}$ that is the identity matrix for the first d rows then zero matrix for the reaming rows
  - so finally lets consider $||X(X^TX)^{-1}X^T - I)\epsilon||_2^2$
    $= ||(A-I)\epsilon||_2^2 = ((A-i)(\epsilon))^t((A-i)(\epsilon)) = \epsilon^tA^TA\epsilon - \epsilon^TA\epsilon - \epsilon^TA\epsilon + \epsilon^t\epsilon$
    $= -\epsilon^TA\epsilon + \epsilon^t\epsilon = \epsilon^T(I-A)\epsilon$
  - so we can write $E\left[\frac{1}{N}\left|\left|\left(X(X^TX)^{-1}X^T - I\right)\epsilon\right|\right|_2^2\right] = E[\frac{1}{N}\epsilon^T(I-A)\epsilon]$
    $= \frac{1}{N}(E[\epsilon^T\epsilon] - E[\epsilon^TA\epsilon]) = \frac{1}{N}(E[\epsilon^T\epsilon] - E[\epsilon]^2 - E[\epsilon^TA\epsilon])$ here as we know each ϵ_i is an random variable and we know that ϵ is an d dimensional vector it must be the case that $\frac{1}{N}(E[\epsilon^T\epsilon] - E[\epsilon]^2 - E[\epsilon^TA\epsilon])$
    $= \frac{1}{N}(E[\Sigma_{i=1}^n\epsilon_i^2] - E[\epsilon]^2 - E[\epsilon^TA\epsilon])$
    $= \frac{1}{N}(\Sigma_{i=1}^nE[\epsilon_i^2 - E[\epsilon]^2] - E[\epsilon^TA\epsilon])$
    $= \frac{1}{N}(E[\Sigma_{i=1}^n\sigma] - E[\epsilon^TA\epsilon])$
    $= \frac{1}{N}(N\sigma^2 - E[\epsilon^TA\epsilon])$ from here we can use eigen decompositon to write $\frac{1}{N}(N\sigma^2 - E[\epsilon^TA\epsilon]) = \frac{1}{N}(N\sigma^2 - E[\epsilon^T(Q^{-1}\Lambda Q\epsilon]) = \frac{1}{N}(N\sigma^2 - (\Sigma_{i=1}^dE[\epsilon_i^2 - E[\epsilon]^2] = \frac{1}{N}(N\sigma^2 - (\Sigma_{i=1}^d\sigma$
    $= \frac{1}{n}(n\sigma - d\sigma)$
    $= \sigma\frac{N-d}{N}$ which was the desired result

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 over fit the data.

2 Q2 **4 / 4**

✓ **- 0 pts** *Correct*

- 2 pts incomplete proof

- 4 pts missing proof

- 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 us define $A = X(X^TX)^{-1}X^T$
  - first we can show that A is symmetric ie $A^t = A$
    - $A^t = (X(X^TX)^{-1}X^T)^T = (X^T)^T((X^TX)^{-1})^TX^T = (X^T)^T((X^TX)^T)^{-1}X^T = (X)((XX^T))^{-1}X^T = A$
  - next we can show that $A = A^TA = A^2$
    - as we showed above A is symmetric we know that $A^2 = A^TA$
    - thus we have $A^2 = A^TA = (X(X^TX)^{-1}X^T)(X(X^TX)^{-1}X^T) = X(X^TX)^{-1}(X^TX)(X^TX)^{-1}X^T = XI(X^TX)^{-1}X^T = X(X^TX)^{-1}X^T = A$
  - now we can reason about the eigenvalues of A
    - suppose $v \in (X)$ this would imply that $\exists u \in \mathbb{R}^{\mathbb{D}} Xu = v$ so consider $Av = AXu = X(X^TX)^{-1}X^TXu = X(X^TX)^{-1}(X^TX)u = X(X^TX)^{-1}X^TXu = Xu = v$ thus any vector $v \in img(X)$ is an eigenvector of A with eigenvalue 1.
    - we know that X is full rank meaning $dim(img(X)) = \mathbb{R}^d$ thus by rank nullity theorem $dim(ker(x)) = dim(ker(x^t)) = \mathbb{R}^{d-n}$
    - we can also see that $A \in \mathbb{R}^{N\times N}$ thus it is clear that A will have d eigenvectors with value 1.
    - we know on the other hand that if $v \in ker(x^t)$ we can write that $X^tv = 0$ thus $Av = X(X^TX)^{-1}X^Tv = 0$ meiang that $v \in ker(A)$ thus we know that A has n-d eigen values of value 0
    - this further implies that the $rank(X) = Rank(A) = d$
    - and finally that the Eigenmatrix of A can be expressed as $\begin{pmatrix} 1 & 0 & ... & 0 \\ 0 & 1.. & 0 & \\ 0 & 0 & ... & 0 \end{pmatrix}$ that is the identity matrix for the first d rows then zero matrix for the reaming rows
  - so finally lets consider $||X(X^TX)^{-1}X^T - I)\epsilon||_2^2$
    $= ||(A-I)\epsilon||_2^2 = ((A-i)(\epsilon))^t((A-i)(\epsilon)) = \epsilon^tA^TA\epsilon - \epsilon^TA\epsilon - \epsilon^TA\epsilon + \epsilon^t\epsilon$
    $= -\epsilon^TA\epsilon + \epsilon^t\epsilon = \epsilon^T(I-A)\epsilon$
  - so we can write $E\left[\frac{1}{N}\left|\left|\left(X(X^TX)^{-1}X^T - I\right)\epsilon\right|\right|_2^2\right] = E[\frac{1}{N}\epsilon^T(I-A)\epsilon]$
    $= \frac{1}{N}(E[\epsilon^T\epsilon] - E[\epsilon^TA\epsilon]) = \frac{1}{N}(E[\epsilon^T\epsilon] - E[\epsilon]^2 - E[\epsilon^TA\epsilon])$ here as we know each ϵ_i is an random variable and we know that ϵ is an d dimensional vector it must be the case that $\frac{1}{N}(E[\epsilon^T\epsilon] - E[\epsilon]^2 - E[\epsilon^TA\epsilon])$
    $= \frac{1}{N}(E[\Sigma_{i=1}^n\epsilon_i^2] - E[\epsilon]^2 - E[\epsilon^TA\epsilon])$
    $= \frac{1}{N}(\Sigma_{i=1}^nE[\epsilon_i^2 - E[\epsilon]^2] - E[\epsilon^TA\epsilon])$
    $= \frac{1}{N}(E[\Sigma_{i=1}^n\sigma] - E[\epsilon^TA\epsilon])$
    $= \frac{1}{N}(N\sigma^2 - E[\epsilon^TA\epsilon])$ from here we can use eigen decompositon to write $\frac{1}{N}(N\sigma^2 - E[\epsilon^TA\epsilon]) = \frac{1}{N}(N\sigma^2 - E[\epsilon^T(Q^{-1}\Lambda Q\epsilon]) = \frac{1}{N}(N\sigma^2 - (\Sigma_{i=1}^dE[\epsilon_i^2 - E[\epsilon]^2] = \frac{1}{N}(N\sigma^2 - (\Sigma_{i=1}^d\sigma$
    $= \frac{1}{n}(n\sigma - d\sigma)$
    $= \sigma\frac{N-d}{N}$ which was the desired result

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 over fit the data.

- as we can see training error $= \sigma\frac{N-d}{N}$ is reducing in d, so naturally raising d while keeping n constant will reduce in a lower training error.
- however this naturally does not mean that our model will generalize better to unseen data despite the lower training error we are getting

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.

   - ```
     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
         train_vals=[i for i in range(len(train[0,:])) if (np.all(train ==
         ↪   train[0,:], axis = 0)[i]==False)]
         test_vals=[i for i in range(len(test[0,:])) if (np.all(test ==
         ↪   test[0,:], axis = 0)[i]==False)]
     ```
     ```

3 Q3 **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts incorrect

- as we can see training error $= \sigma\frac{N-d}{N}$ is reducing in d, so naturally raising d while keeping n constant will reduce in a lower training error.
- however this naturally does not mean that our model will generalize better to unseen data despite the lower training error we are getting

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.

   - ```
     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
         train_vals=[i for i in range(len(train[0,:])) if (np.all(train ==
         ↪   train[0,:], axis = 0)[i]==False)]
         test_vals=[i for i in range(len(test[0,:])) if (np.all(test ==
         ↪   test[0,:], axis = 0)[i]==False)]
     ```
     ```

```
## standardizes across non_constant rows. and discards constnat
↪  features.
train_numerator=(train[:,train_vals]-
↪  np.min(train[:,train_vals],axis=0))
train_denominator=(np.max(train[:,train_vals],axis=0)-
↪  np.min(train[:,train_vals],axis=0))
test_numeroatro=(test[:,test_vals]-
↪  np.min(test[:,test_vals],axis=0))
test_denomiator=(np.max(test[:,test_vals],axis=0)-
↪  np.min(test[:,test_vals],axis=0))
return train_numerator/train_denominator,
↪  test_numeroatro/test_denomiator
```

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 $y = (y_1, \ldots, y_m)^T \in \mathbb{R}^{m \times 1}$

   - the expression $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\boldsymbol{x}_i) - y_i \right)^2$ can be expressed in matrix form as $J(\theta) = \frac{1}{m} ||X\theta - y||_2^2$ I am not sure how much justification is required here since we showed this on homework one question 5 effectively

   be the *response*. Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign. [1]

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

[1] Being able to write expressions as matrix/vector expressions without summations is crucial to making implementations that are useful in practice, since you can use numpy (or more generally, an efficient numerical linear algebra library) to implement these matrix/vector operations orders of magnitude faster than naively implementing with loops in Python.

4 Q4 0 / 1

- 0 pts Correct

✓ **- 1 pts** *incorrect implementation, pay attention to whether you used train max&min to normalize and whether the result normalizes to [0, 1]*

- 0.5 pts need to discard features that are constant in the training set before normalization

```
## standardizes across non_constant rows. and discards constnat
↪  features.
train_numerator=(train[:,train_vals]-
↪  np.min(train[:,train_vals],axis=0))
train_denominator=(np.max(train[:,train_vals],axis=0)-
↪  np.min(train[:,train_vals],axis=0))
test_numeroatro=(test[:,test_vals]-
↪  np.min(test[:,test_vals],axis=0))
test_denomiator=(np.max(test[:,test_vals],axis=0)-
↪  np.min(test[:,test_vals],axis=0))
return train_numerator/train_denominator,
↪  test_numeroatro/test_denomiator
```

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 $y = (y_1, \ldots, y_m)^T \in \mathbb{R}^{m \times 1}$

   - the expression $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\boldsymbol{x}_i) - y_i \right)^2$ can be expressed in matrix form as $J(\theta) = \frac{1}{m} ||X\theta - y||_2^2$ I am not sure how much justification is required here since we showed this on homework one question 5 effectively

   be the *response*. Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign. [1]

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

[1] Being able to write expressions as matrix/vector expressions without summations is crucial to making implementations that are useful in practice, since you can use numpy (or more generally, an efficient numerical linear algebra library) to implement these matrix/vector operations orders of magnitude faster than naively implementing with loops in Python.

- from last problem we can see that our objective function is $J(\theta) = \frac{1}{m}||X\theta - y||_2^2$
- we can expand this as $J(\theta) = \frac{1}{m}||X\theta - y||_2^2 = \frac{1}{m}(X\theta - y)^T(X\theta - y)$
  $= \frac{1}{m}(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^t y)$
- then as differentiation is linear we can take the derivative of each part independently
- $\frac{\partial \theta^T X^T X\theta}{\partial \theta}$
  - we can think of $\theta^T X^T X\theta$ as $\theta^T A\theta$ where $A = X^T X$ further we can see $A^t = (X^T X)^T = X^T X = A$ thus as is symmetric and $\frac{\partial \theta^T X^T X\theta}{\partial \theta} = 2\theta(A) = 2\theta^t(X^T X)$
- $\frac{\partial \theta^T X^T y}{\partial \theta}$
  - we can express $\theta^T X^T y = \alpha$ for some $\alpha \in \mathbb{R}$ as $\theta^T v = \alpha$ where $v = X^T y$ thus we see that $\frac{\partial \theta^T X^T y}{\partial \theta} = \frac{\partial \theta^T v}{\partial \theta} = v^t = (X^T y)^T = X^T y$
- $\frac{\partial y^T X\theta}{\partial \theta}$
  - a$\frac{\partial y^T X\theta}{\partial \theta} = \alpha$ for some constnat
  - we can write $w^t = y^T X$ and then see that $\alpha = \frac{\partial y^T X\theta}{\partial \theta} = \frac{\partial w^T \theta}{\partial \theta} = w^T = Xy^T$
- so finally in sum we see that $\nabla_J(\theta) = \frac{1}{m}(2(X^t X)\theta - 2X^T y) = \frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y)$

7. Write down the expression for updating θ in the gradient descent algorithm for a step size η.
   - $\theta = \theta - \eta\nabla_J(\theta) = \theta - \eta(\frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y))$
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.
   - ```
     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
         return np.dot((X@theta-y),(X@theta-y))/y.shape[0]
     ```
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.
     ```

5 Q5 **1 / 1**

✓ **- 0 pts** *Correct*

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

```
## standardizes across non_constant rows. and discards constnat
↪  features.
train_numerator=(train[:,train_vals]-
↪  np.min(train[:,train_vals],axis=0))
train_denominator=(np.max(train[:,train_vals],axis=0)-
↪  np.min(train[:,train_vals],axis=0))
test_numeroatro=(test[:,test_vals]-
↪  np.min(test[:,test_vals],axis=0))
test_denomiator=(np.max(test[:,test_vals],axis=0)-
↪  np.min(test[:,test_vals],axis=0))
return train_numerator/train_denominator,
↪  test_numeroatro/test_denomiator
```

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 $y = (y_1, \ldots, y_m)^T \in \mathbb{R}^{m \times 1}$

   - the expression $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\boldsymbol{x}_i) - y_i \right)^2$ can be expressed in matrix form as $J(\theta) = \frac{1}{m} ||X\theta - y||_2^2$ I am not sure how much justification is required here since we showed this on homework one question 5 effectively

   be the *response*. Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign. [1]

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

[1] Being able to write expressions as matrix/vector expressions without summations is crucial to making implementations that are useful in practice, since you can use numpy (or more generally, an efficient numerical linear algebra library) to implement these matrix/vector operations orders of magnitude faster than naively implementing with loops in Python.

- from last problem we can see that our objective function is $J(\theta) = \frac{1}{m}||X\theta - y||_2^2$
- we can expand this as $J(\theta) = \frac{1}{m}||X\theta - y||_2^2 = \frac{1}{m}(X\theta - y)^T(X\theta - y)$
  $= \frac{1}{m}(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^t y)$
- then as differentiation is linear we can take the derivative of each part independently
- $\frac{\partial \theta^T X^T X\theta}{\partial \theta}$
  - we can think of $\theta^T X^T X\theta$ as $\theta^T A\theta$ where $A = X^T X$ further we can see $A^t = (X^T X)^T = X^T X = A$ thus as is symmetric and $\frac{\partial \theta^T X^T X\theta}{\partial \theta} = 2\theta(A) = 2\theta^t(X^T X)$
- $\frac{\partial \theta^T X^T y}{\partial \theta}$
  - we can express $\theta^T X^T y = \alpha$ for some $\alpha \in \mathbb{R}$ as $\theta^T v = \alpha$ where $v = X^T y$ thus we see that $\frac{\partial \theta^T X^T y}{\partial \theta} = \frac{\partial \theta^T v}{\partial \theta} = v^t = (X^T y)^T = X^T y$
- $\frac{\partial y^T X\theta}{\partial \theta}$
  - a$\frac{\partial y^T X\theta}{\partial \theta} = \alpha$ for some constnat
  - we can write $w^t = y^T X$ and then see that $\alpha = \frac{\partial y^T X\theta}{\partial \theta} = \frac{\partial w^T \theta}{\partial \theta} = w^T = Xy^T$
- so finally in sum we see that $\nabla_J(\theta) = \frac{1}{m}(2(X^t X)\theta - 2X^T y) = \frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y)$

7. Write down the expression for updating θ in the gradient descent algorithm for a step size η.
   - $\theta = \theta - \eta\nabla_J(\theta) = \theta - \eta(\frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y))$
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.
   - ```
     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
         return np.dot((X@theta-y),(X@theta-y))/y.shape[0]
     ```
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.
     ```

6 Q6 1 / 1

✓ **- 0 pts** *Correct*

- 0 pts Consistent with previous expression

- 0.5 pts $$\nabla J = \frac{2}{m}X^\top (X\theta - y)$$

- 0.5 pts Transposed result; the gradient of $$J$$ is a column vector

- from last problem we can see that our objective function is $J(\theta) = \frac{1}{m}||X\theta - y||_2^2$
- we can expand this as $J(\theta) = \frac{1}{m}||X\theta - y||_2^2 = \frac{1}{m}(X\theta - y)^T(X\theta - y)$
  $= \frac{1}{m}(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^t y)$
- then as differentiation is linear we can take the derivative of each part independently
- $\frac{\partial \theta^T X^T X\theta}{\partial \theta}$
  - we can think of $\theta^T X^T X\theta$ as $\theta^T A\theta$ where $A = X^T X$ further we can see $A^t = (X^T X)^T = X^T X = A$ thus as is symmetric and $\frac{\partial \theta^T X^T X\theta}{\partial \theta} = 2\theta(A) = 2\theta^t(X^T X)$
- $\frac{\partial \theta^T X^T y}{\partial \theta}$
  - we can express $\theta^T X^T y = \alpha$ for some $\alpha \in \mathbb{R}$ as $\theta^T v = \alpha$ where $v = X^T y$ thus we see that $\frac{\partial \theta^T X^T y}{\partial \theta} = \frac{\partial \theta^T v}{\partial \theta} = v^t = (X^T y)^T = X^T y$
- $\frac{\partial y^T X\theta}{\partial \theta}$
  - a$\frac{\partial y^T X\theta}{\partial \theta} = \alpha$ for some constnat
  - we can write $w^t = y^T X$ and then see that $\alpha = \frac{\partial y^T X\theta}{\partial \theta} = \frac{\partial w^T \theta}{\partial \theta} = w^T = Xy^T$
- so finally in sum we see that $\nabla_J(\theta) = \frac{1}{m}(2(X^t X)\theta - 2X^T y) = \frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y)$

7. Write down the expression for updating θ in the gradient descent algorithm for a step size η.
   - $\theta = \theta - \eta\nabla_J(\theta) = \theta - \eta(\frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y))$
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.
   - ```
     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
         return np.dot((X@theta-y),(X@theta-y))/y.shape[0]
     ```
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.
     ```

7 Q7 1 / 1

✓ **- 0 pts** *Correct*

- 1 pts Update is done on $$\theta$$

- 1 pts Update needs to go at the opposite direction of the gradient

- from last problem we can see that our objective function is $J(\theta) = \frac{1}{m}||X\theta - y||_2^2$
- we can expand this as $J(\theta) = \frac{1}{m}||X\theta - y||_2^2 = \frac{1}{m}(X\theta - y)^T(X\theta - y)$
  $= \frac{1}{m}(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^t y)$
- then as differentiation is linear we can take the derivative of each part independently
- $\frac{\partial \theta^T X^T X\theta}{\partial \theta}$
  - we can think of $\theta^T X^T X\theta$ as $\theta^T A\theta$ where $A = X^T X$ further we can see $A^t = (X^T X)^T = X^T X = A$ thus as is symmetric and $\frac{\partial \theta^T X^T X\theta}{\partial \theta} = 2\theta(A) = 2\theta^t(X^T X)$
- $\frac{\partial \theta^T X^T y}{\partial \theta}$
  - we can express $\theta^T X^T y = \alpha$ for some $\alpha \in \mathbb{R}$ as $\theta^T v = \alpha$ where $v = X^T y$ thus we see that $\frac{\partial \theta^T X^T y}{\partial \theta} = \frac{\partial \theta^T v}{\partial \theta} = v^t = (X^T y)^T = X^T y$
- $\frac{\partial y^T X\theta}{\partial \theta}$
  - a$\frac{\partial y^T X\theta}{\partial \theta} = \alpha$ for some constnat
  - we can write $w^t = y^T X$ and then see that $\alpha = \frac{\partial y^T X\theta}{\partial \theta} = \frac{\partial w^T \theta}{\partial \theta} = w^T = Xy^T$
- so finally in sum we see that $\nabla_J(\theta) = \frac{1}{m}(2(X^t X)\theta - 2X^T y) = \frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y)$

7. Write down the expression for updating θ in the gradient descent algorithm for a step size η.
   - $\theta = \theta - \eta\nabla_J(\theta) = \theta - \eta(\frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y))$
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.
   - ```
     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
         return np.dot((X@theta-y),(X@theta-y))/y.shape[0]
     ```
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.
     ```

8 Q8 **1 / 1**

✓ **- 0 pts** *Correct*

- 0 pts Consistent with previous result

- 1 pts Missing submission

- from last problem we can see that our objective function is $J(\theta) = \frac{1}{m}||X\theta - y||_2^2$
- we can expand this as $J(\theta) = \frac{1}{m}||X\theta - y||_2^2 = \frac{1}{m}(X\theta - y)^T(X\theta - y)$
  $= \frac{1}{m}(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^t y)$
- then as differentiation is linear we can take the derivative of each part independently
- $\frac{\partial \theta^T X^T X\theta}{\partial \theta}$
  - we can think of $\theta^T X^T X\theta$ as $\theta^T A\theta$ where $A = X^T X$ further we can see $A^t = (X^T X)^T = X^T X = A$ thus as is symmetric and $\frac{\partial \theta^T X^T X\theta}{\partial \theta} = 2\theta(A) = 2\theta^t(X^T X)$
- $\frac{\partial \theta^T X^T y}{\partial \theta}$
  - we can express $\theta^T X^T y = \alpha$ for some $\alpha \in \mathbb{R}$ as $\theta^T v = \alpha$ where $v = X^T y$ thus we see that $\frac{\partial \theta^T X^T y}{\partial \theta} = \frac{\partial \theta^T v}{\partial \theta} = v^t = (X^T y)^T = X^T y$
- $\frac{\partial y^T X\theta}{\partial \theta}$
  - a$\frac{\partial y^T X\theta}{\partial \theta} = \alpha$ for some constnat
  - we can write $w^t = y^T X$ and then see that $\alpha = \frac{\partial y^T X\theta}{\partial \theta} = \frac{\partial w^T \theta}{\partial \theta} = w^T = Xy^T$
- so finally in sum we see that $\nabla_J(\theta) = \frac{1}{m}(2(X^t X)\theta - 2X^T y) = \frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y)$

7. Write down the expression for updating θ in the gradient descent algorithm for a step size η.
   - $\theta = \theta - \eta\nabla_J(\theta) = \theta - \eta(\frac{2}{m}((X^t X)\theta - X^t y) = \frac{2}{m}X^t(X\theta - y))$
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.
   - ```
     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
         return np.dot((X@theta-y),(X@theta-y))/y.shape[0]
     ```
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.
     ```

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

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.

    - ```
      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:
      ```

9 Q9 **1 / 1**

✓ **- 0 pts** *Correct*

- 0 pts Consistent

- 0.5 pts Factor 2 missing

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

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.

    - ```
      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= list(map(lambda I:( compute_square_loss(X,y,theta +
        ↪   epsilon * I)-compute_square_loss(X,y,theta - epsilon * I)
        ↪   )/(2*epsilon),np.identity(num_features)))
        distances=true_gradient-approx_grad
        euclidian_distance=(distances.T)@(distances)
        return euclidian_distance<=tolerance


    ########################################
    ### Generic gradient checker
    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_square_loss_gradient
        """
        true_gradient = gradient_func(X, y, theta) #The true gradient
```

```
    #print("true gradinet is ", true_gradient)
    num_features = theta.shape[0]
    # approx_grad = np.zeros(num_features) #Initialize the gradient we
    ↪  approximate
    approx_grad= list(map(lambda I:( objective_func(X,y,theta +
    ↪  epsilon * I)-objective_func(X,y,theta - epsilon * I)
    ↪  )/(2*epsilon),np.identity(num_features)))
    #print("print estimated gradient is", approx_grad)
    distances=true_gradient-approx_grad
    euclidian_distance=(distances.T)@(distances)
    #print("there euclidian distance is, ", euclidian_distance)
    result=euclidian_distance<=tolerance
    #print("thus is is {0} that the true and aproximate gradients are
    ↪  within tollerence".format(result))
    return euclidian_distance<=tolerance
```

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.

```
• def batch_grad_descent(X, y, alpha=0.1, num_step=1000,
  ↪  grad_check=False):
    """
    In this question you will implement batch gradient descent to
    minimize the average square loss objective.

    Args:
        X - the feature vector, 2D numpy array of size(num_instances,
  ↪  num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        alpha - step size in gradient descent
        num_step - number of steps to run
        grad_check - a boolean value indicating whether checking the
  ↪  gradient when updating

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array
  ↪  of size(num_step+1, num_features)
                    for instance, theta in step 0 should be
  ↪  theta_hist[0], theta in step(num_step) is theta_hist[-1]
        loss_hist - the history of average square loss on the data, 1D
  ↪  numpy array,(num_step+1)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step + 1, num_features))  #Initialize
    ↪  theta_hist
```

10 Q10 **1.5 / 2**

- **0 pts** Correct
- **0.5 pts** Should use the norm or element wise
- **0.5 pts** If using the norm, should take the norm of the difference

✓ **- 0.5 pts** *Missing a square root for the distance*

```
    #print("true gradinet is ", true_gradient)
    num_features = theta.shape[0]
    # approx_grad = np.zeros(num_features) #Initialize the gradient we
    ↪  approximate
    approx_grad= list(map(lambda I:( objective_func(X,y,theta +
    ↪  epsilon * I)-objective_func(X,y,theta - epsilon * I)
    ↪  )/(2*epsilon),np.identity(num_features)))
    #print("print estimated gradient is", approx_grad)
    distances=true_gradient-approx_grad
    euclidian_distance=(distances.T)@(distances)
    #print("there euclidian distance is, ", euclidian_distance)
    result=euclidian_distance<=tolerance
    #print("thus is is {0} that the true and aproximate gradients are
    ↪  within tollerence".format(result))
    return euclidian_distance<=tolerance
```

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.

```
• def batch_grad_descent(X, y, alpha=0.1, num_step=1000,
  ↪  grad_check=False):
    """
    In this question you will implement batch gradient descent to
    minimize the average square loss objective.

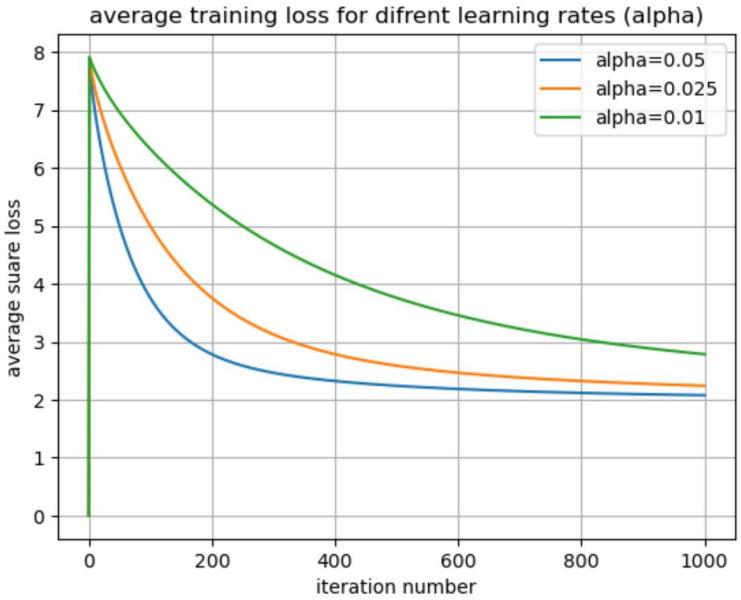
    Args:
        X - the feature vector, 2D numpy array of size(num_instances,
  ↪  num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        alpha - step size in gradient descent
        num_step - number of steps to run
        grad_check - a boolean value indicating whether checking the
  ↪  gradient when updating

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array
  ↪  of size(num_step+1, num_features)
                    for instance, theta in step 0 should be
  ↪  theta_hist[0], theta in step(num_step) is theta_hist[-1]
        loss_hist - the history of average square loss on the data, 1D
  ↪  numpy array,(num_step+1)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step + 1, num_features))  #Initialize
    ↪  theta_hist
```

```
loss_hist = np.zeros(num_step + 1)  #Initialize loss_hist
theta = np.zeros(num_features)  #Initialize theta
i=0
while i<num_step:
    current_grad=compute_square_loss_gradient(X, y, theta)
    theta=theta-(alpha*current_grad)
    theta_hist[i+1]=theta
    loss_hist[i+1]=compute_square_loss(X,y,theta)
    i=i+1
return theta_hist,loss_hist
```

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.

    - it seems that learning rates bellow .75 are able to converge.
    - .05 seems to be able to converge the fastest this is what we would expect as it is the largest learning rate that seems to converge.
    - the values above .75 seem to have issues with numeric stability, but clearly the average loss is going up
    - here are my graphs

11 Q11 **0.5 / 1**

- **0 pts** Correct

✓ **- 0.5 pts** *Partially functioning code*

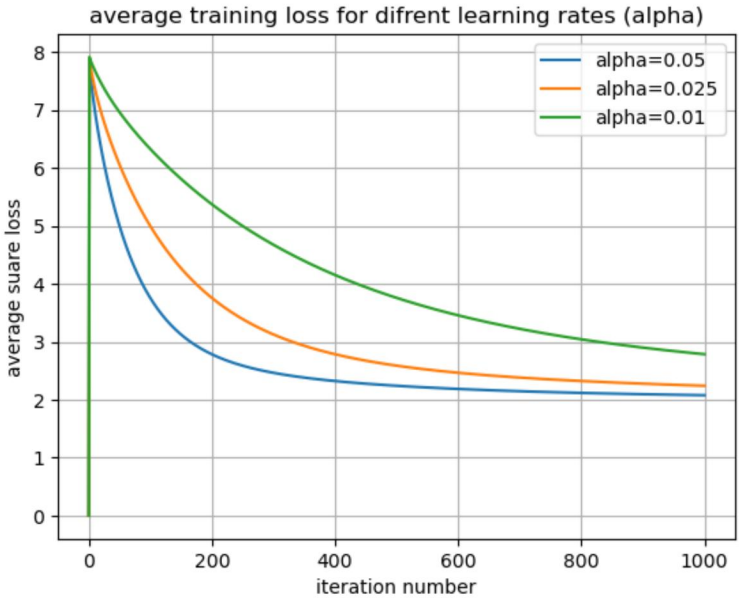
- **1 pts** Code not functioning
- **0.25 pts** Other minor issue

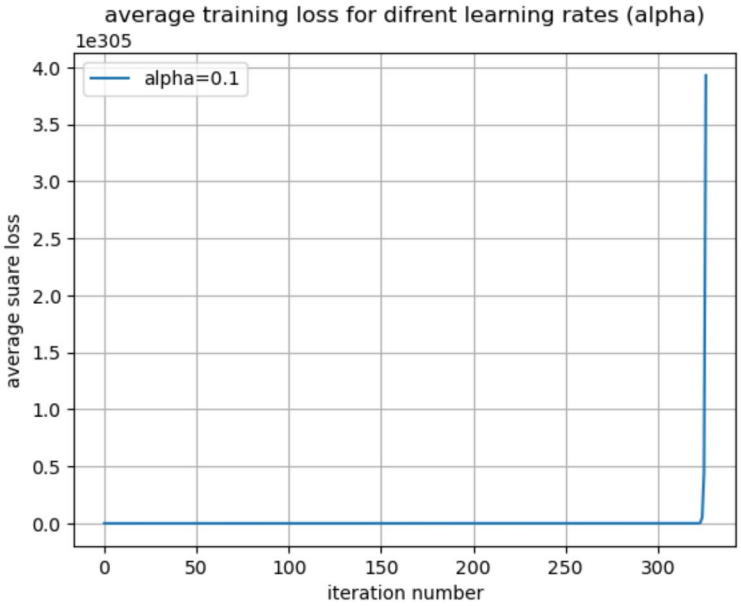
missing grad check

```
loss_hist = np.zeros(num_step + 1)  #Initialize loss_hist
theta = np.zeros(num_features)  #Initialize theta
i=0
while i<num_step:
    current_grad=compute_square_loss_gradient(X, y, theta)
    theta=theta-(alpha*current_grad)
    theta_hist[i+1]=theta
    loss_hist[i+1]=compute_square_loss(X,y,theta)
    i=i+1
return theta_hist,loss_hist
```

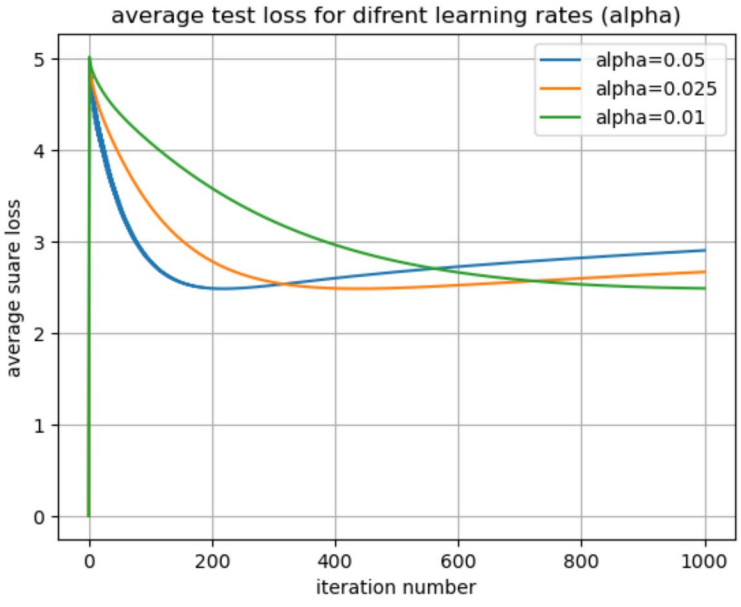
12. Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge. Starting with a step-size of 0.1, try various different fixed step sizes to see which converges most quickly and/or which diverge. As a minimum, try step sizes 0.5, 0.1, .05, and .01. 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.

    - it seems that learning rates bellow .75 are able to converge.
    - .05 seems to be able to converge the fastest this is what we would expect as it is the largest learning rate that seems to converge.
    - the values above .75 seem to have issues with numeric stability, but clearly the average loss is going up
    - here are my graphs



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.



- it seems the average learning rate in all values ofα where we saw convergence law problem see decreases in test loss for a certain number of iterations and then increases after that which indicates over fitting

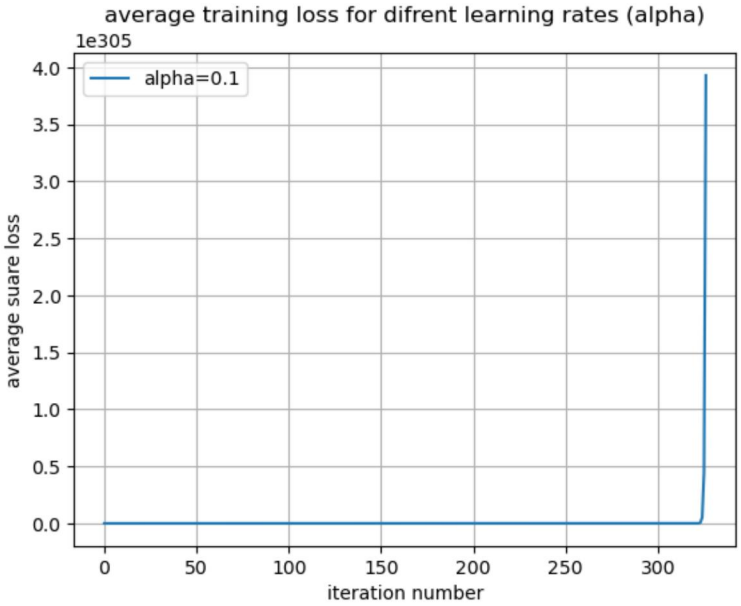
12 Q12 **0.75 / 1**

- **0 pts** Correct
- **0.5 pts** No plot or wrong plot
- **0.5 pts** No or wrong findings

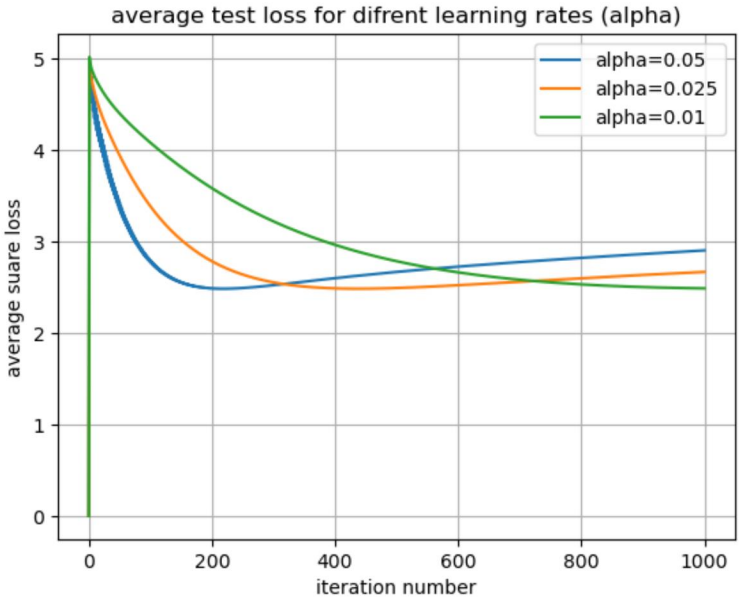
✓ **- 0.25 pts** *Click here to replace this description.*

- **0.5 pts** Click here to replace this description.

missing 0.5 case required by the question.



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.



- it seems the average learning rate in all values ofα where we saw convergence law problem see decreases in test loss for a certain number of iterations and then increases after that which indicates over fitting

13 Q13 **1 / 1**

✓ **- 0 pts** *please plot for the only one learning rate you selected from Q12*

- 0 pts Correct

- 1 pts wrong plot. you should observe loss first decrease and then increase. the plot should also be smooth

- 0.5 pts Click here to replace this description.

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)

    - $J_\lambda(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(\boldsymbol{x}_i) - y_i\right)^2 + \lambda\theta^T\theta = \frac{1}{m}||X\theta - y||_2^2 + \lambda||\theta||_2^2$
    - thus we can see that our gradient of ridge regression is $\nabla(J_\lambda(\theta)) = \frac{2}{m}X^t(X\theta - y) + 2\lambda\theta$
    - thus our update rule becomes $\theta = \theta - \eta\nabla(J_\lambda(\theta)) = \theta - \eta\nabla(\frac{2}{m}X^t(X\theta - y) + 2\lambda\theta)$

15. Implement `compute_regularized_square_loss_gradient`.

    - ```
      def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
          """
          Compute the gradient of L2-regularized average square loss
      ↪   function given X, y and theta

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

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

16. Implement `regularized_grad_descent`.

    - ```
      def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2,
      ↪   num_step=1000):
          """
          Args:
      ```

14 Q14 **0.5 / 1**

- **0 pts** Correct
- **0.5 pts** Wrong or missing gradient

✓ **- 0.5 pts** *Wrong or missing theta update expression*

you don't need to take gradient again in your last expression

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)

    - $J_\lambda(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(\boldsymbol{x}_i) - y_i\right)^2 + \lambda\theta^T\theta = \frac{1}{m}||X\theta - y||_2^2 + \lambda||\theta||_2^2$
    - thus we can see that our gradient of ridge regression is $\nabla(J_\lambda(\theta)) = \frac{2}{m}X^t(X\theta - y) + 2\lambda\theta$
    - thus our update rule becomes $\theta = \theta - \eta\nabla(J_\lambda(\theta)) = \theta - \eta\nabla(\frac{2}{m}X^t(X\theta - y) + 2\lambda\theta)$

15. Implement `compute_regularized_square_loss_gradient`.

    - ```
      def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
          """
          Compute the gradient of L2-regularized average square loss
      ↪   function given X, y and theta

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

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

16. Implement `regularized_grad_descent`.

    - ```
      def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2,
      ↪   num_step=1000):
          """
          Args:
      ```

```
        X - the feature vector, 2D numpy array of size(num_instances,
↪  num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        alpha - step size in gradient descent
        lambda_reg - the regularization coefficient
        num_step - number of steps to run

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array
↪  of size(num_step+1, num_features)
                    for instance, theta in step 0 should be
↪  theta_hist[0], theta in step(num_step+1) is theta_hist[-1]
        loss hist - the history of average square loss function
↪  without the regularization term, 1D numpy array.
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_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
    i=0
    while i<num_step:
        current_grad=compute_regularized_square_loss_gradient(X, y,
        ↪  theta,lambda_reg)
        theta=theta-(alpha*current_grad)
        theta_hist[i+1]=theta
        loss_hist[i+1]=compute_square_loss(X,y,theta)
        i=i+1
    return theta_hist,loss_hist
```

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

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?

15 Q15 **1 / 1**

✓ **- 0 pts** *Correct*

- 0.5 pts Partially functioning

- 1 pts Not functioning

- 0 pts Inefficient solution. Do matrix-vector operation instead. remember X.T is transpose of X

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)

    - $J_\lambda(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(\boldsymbol{x}_i) - y_i\right)^2 + \lambda\theta^T\theta = \frac{1}{m}||X\theta - y||_2^2 + \lambda||\theta||_2^2$
    - thus we can see that our gradient of ridge regression is $\nabla(J_\lambda(\theta)) = \frac{2}{m}X^t(X\theta - y) + 2\lambda\theta$
    - thus our update rule becomes $\theta = \theta - \eta\nabla(J_\lambda(\theta)) = \theta - \eta\nabla(\frac{2}{m}X^t(X\theta - y) + 2\lambda\theta)$

15. Implement `compute_regularized_square_loss_gradient`.

    - ```
      def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
          """
          Compute the gradient of L2-regularized average square loss
      ↪   function given X, y and theta

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

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

16. Implement `regularized_grad_descent`.

    - ```
      def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2,
      ↪   num_step=1000):
          """
          Args:
      ```

```
        X - the feature vector, 2D numpy array of size(num_instances,
↪  num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        alpha - step size in gradient descent
        lambda_reg - the regularization coefficient
        num_step - number of steps to run

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array
↪  of size(num_step+1, num_features)
                    for instance, theta in step 0 should be
↪  theta_hist[0], theta in step(num_step+1) is theta_hist[-1]
        loss hist - the history of average square loss function
↪  without the regularization term, 1D numpy array.
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_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
    i=0
    while i<num_step:
        current_grad=compute_regularized_square_loss_gradient(X, y,
        ↪  theta,lambda_reg)
        theta=theta-(alpha*current_grad)
        theta_hist[i+1]=theta
        loss_hist[i+1]=compute_square_loss(X,y,theta)
        i=i+1
    return theta_hist,loss_hist
```

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

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?

16 Q16 **1 / 2**

- **0 pts** Correct
- **0 pts** you can ignore grad_check for this question
- **0 pts** you can use previous functions directly

✓ **- 1 pts** *Partially functioning*

- **2 pts** Not functioning

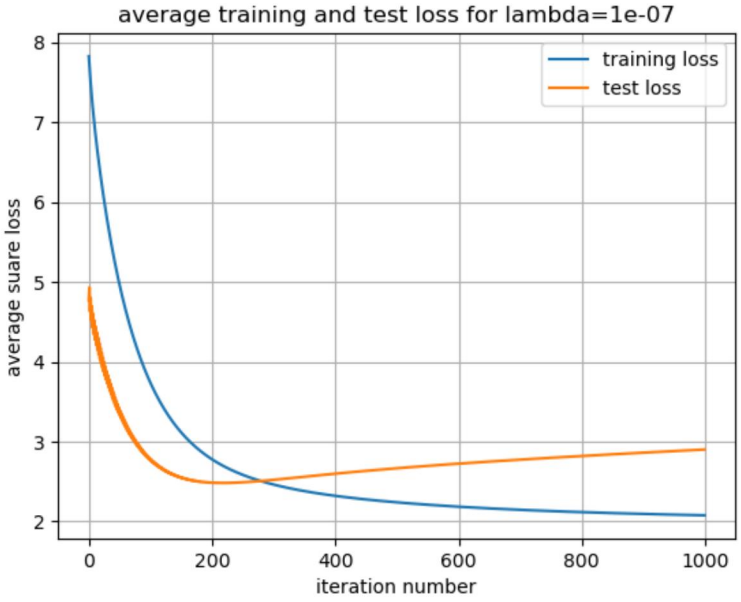
missing initial loss

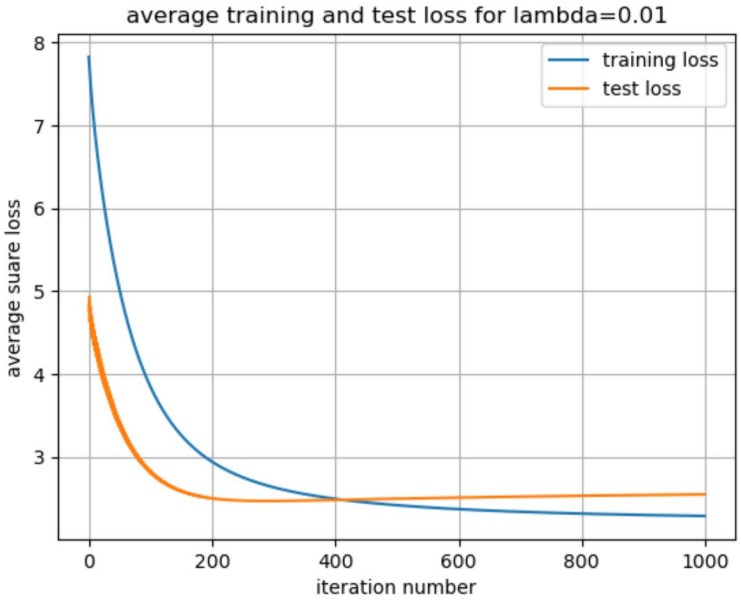
```
        X - the feature vector, 2D numpy array of size(num_instances,
↪  num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        alpha - step size in gradient descent
        lambda_reg - the regularization coefficient
        num_step - number of steps to run

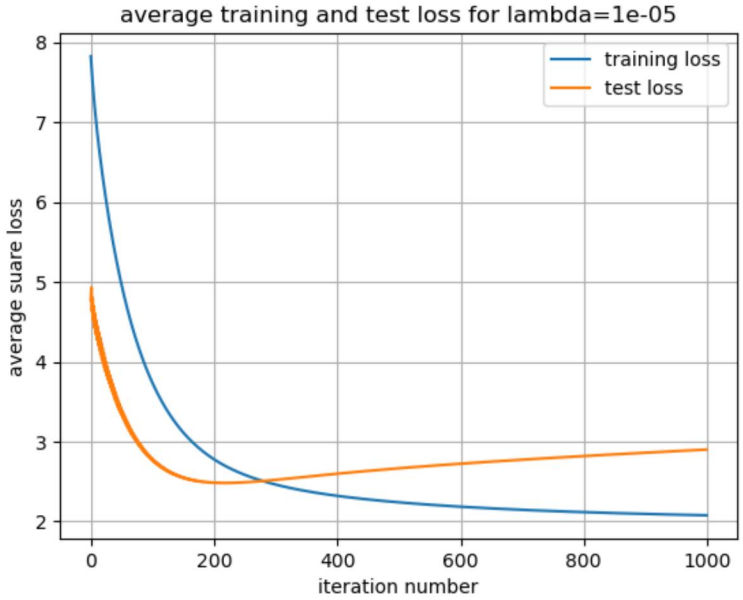
    Returns:
        theta_hist - the history of parameter vector, 2D numpy array
↪  of size(num_step+1, num_features)
                    for instance, theta in step 0 should be
↪  theta_hist[0], theta in step(num_step+1) is theta_hist[-1]
        loss hist - the history of average square loss function
↪  without the regularization term, 1D numpy array.
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_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
    i=0
    while i<num_step:
        current_grad=compute_regularized_square_loss_gradient(X, y,
        ↪  theta,lambda_reg)
        theta=theta-(alpha*current_grad)
        theta_hist[i+1]=theta
        loss_hist[i+1]=compute_square_loss(X,y,theta)
        i=i+1
    return theta_hist,loss_hist
```

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

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?

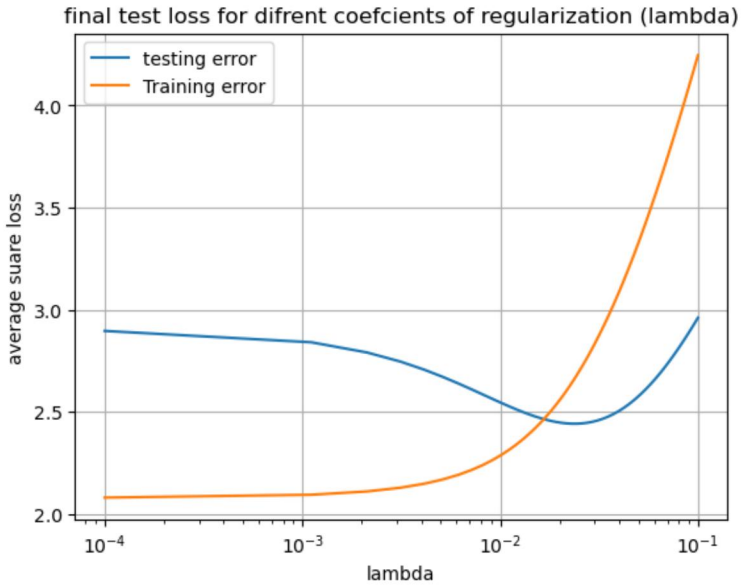

average training and test loss for lambda=1e-07
training loss
test loss
average suare loss
iteration number
0
200
400
600
800
1000
2
3
4
5
6
7
8


average training and test loss for lambda=0.01
training loss
test loss
average suare loss
iteration number
0
200
400
600
800
1000
3
4
5
6
7
8



- looking at these charts it seems like as λ grows the training and lest curve cross earlier meaning that our function is overfititng, more quickly with less training iterations

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 ?



- this graph would seem to suggest a λ around 10^{-2}

19. Another heuristic of regularization is to *early-stop* the training when the test error reaches

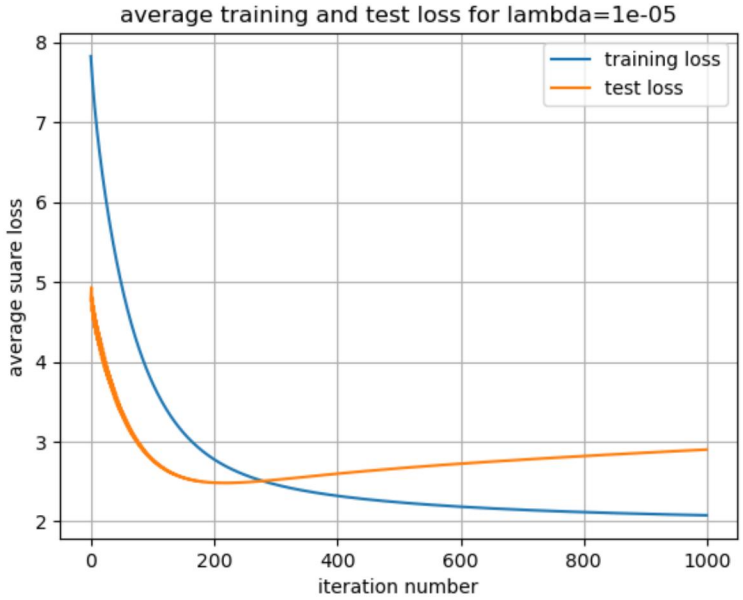
17 Q17 **1 / 1**

✓ **- 0 pts** *Correct*

- 0.5 pts Wrong plots

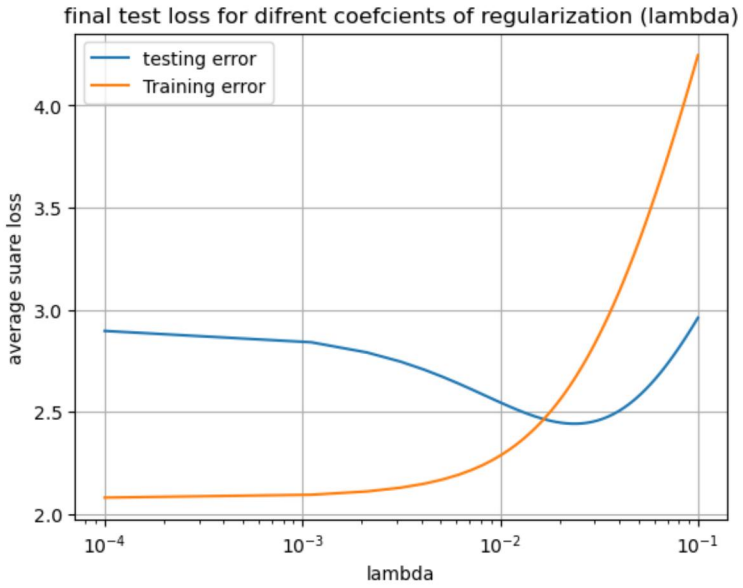
- 0.5 pts Missing discussion

- 1 pts empty/not readable



- looking at these charts it seems like as λ grows the training and lest curve cross earlier meaning that our function is overfititng, more quickly with less training iterations

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 ?



- this graph would seem to suggest a λ around 10^{-2}

19. Another heuristic of regularization is to *early-stop* the training when the test error reaches

18 Q18 **2 / 2**

✓ **- 0 pts** *Correct*

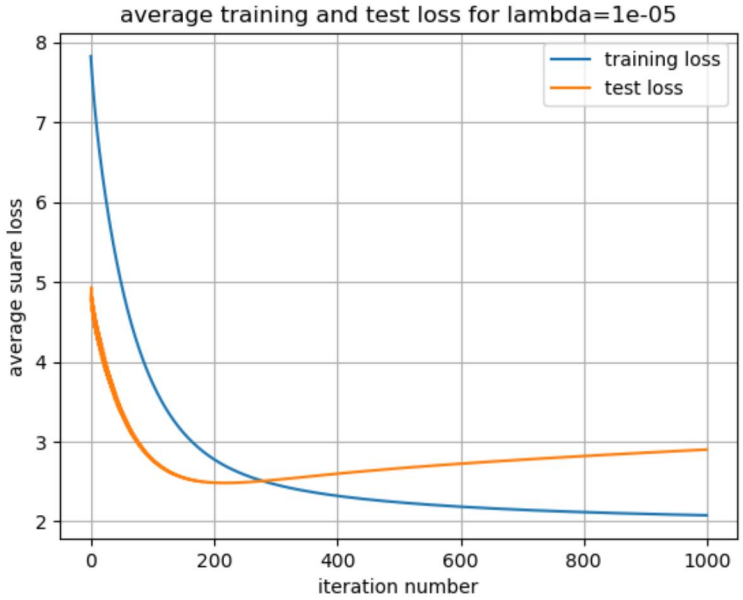
- 1 pts Missing plots

- 1 pts Missing best lambda

- 0.5 pts The best lambda should be around 0.02

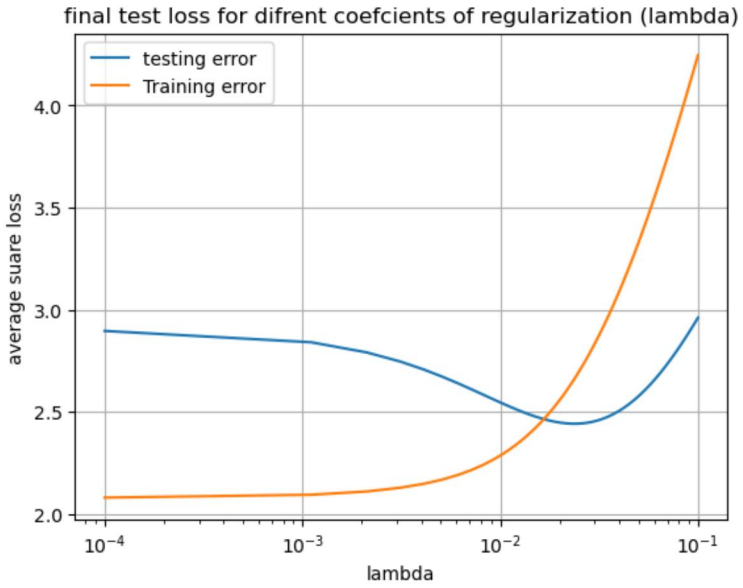
- 1 pts should select lambda based on testing loss

- 2 pts empty/not readable



- looking at these charts it seems like as λ grows the training and lest curve cross earlier meaning that our function is overfititng, more quickly with less training iterations

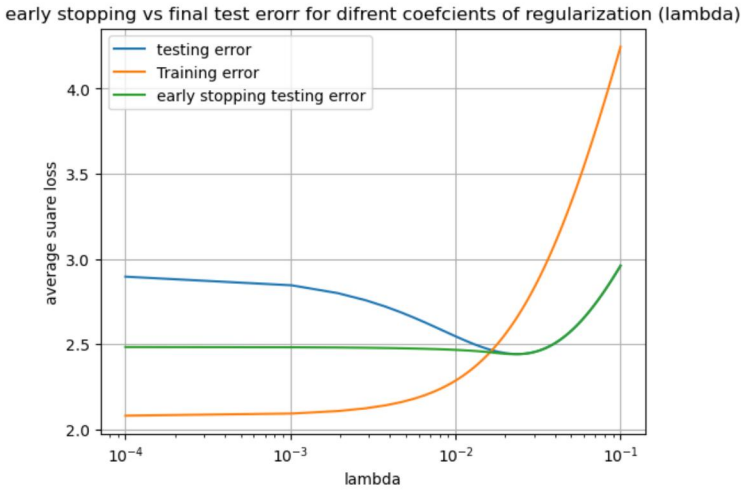
18. Plot the training average square loss and the test average square loss at the end of training as a function of λ. You may want to have $\log(\lambda)$ on the x-axis rather than λ. Which value of λ would you choose ?



- this graph would seem to suggest a λ around 10^{-2}

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?



- they seem to suggest around the same amount. it is imporanat to note however that early stopping makes a very large differnnce for smaller values of λ

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

    - we have limited training data namely or X_train is only 100 individuals, thus we need to be very care full about over fitting our training data. This makes regularization seem important
    - further we can see there are 49 features in our data set. This means we are trying to learn many features from little data, thus i would go with lasso regression, as it seems prudent in a case with so many features and such little data to try to learn the important features well, while avoiding over fitting to the training data.

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.

19 Q19 **1 / 1**

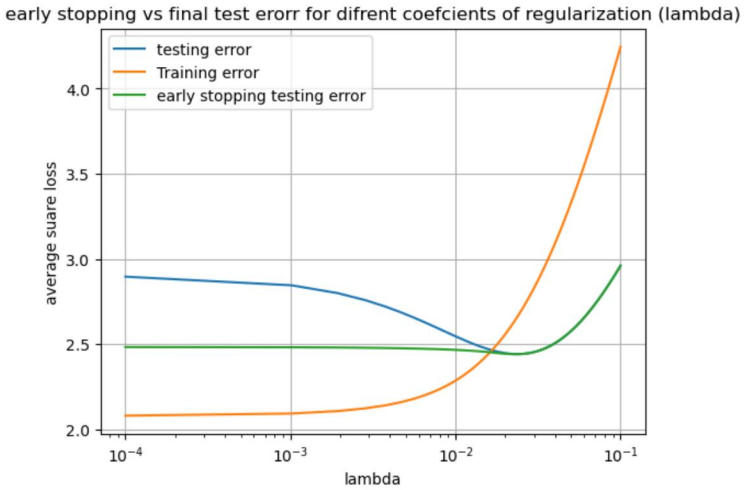
✓ **- 0 pts** *Correct*

- 0.5 pts Missing early stopping curve

- 0.5 pts Missing lambda

- 1 pts empty/not readable

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?



- they seem to suggest around the same amount. it is imporanat to note however that early stopping makes a very large differnnce for smaller values of λ

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

    - we have limited training data namely or X_train is only 100 individuals, thus we need to be very care full about over fitting our training data. This makes regularization seem important
    - further we can see there are 49 features in our data set. This means we are trying to learn many features from little data, thus i would go with lasso regression, as it seems prudent in a case with so many features and such little data to try to learn the important features well, while avoiding over fitting to the training data.

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.

20 Q20 **0.5 / 1**

- **0 pts** Correct

✓ **- 0.5 pts** *Incomplete*

- **0.5 pts** Missing reason
- **0.5 pts** should not select theta based on training error
- **1 pts** should select theta that minimize test error
- **1 pts** empty/not readable

should specify it is the testing loss that we want to minimize

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.

- it is frailly obvious that using $f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta$ we can maintain equality

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

- the expression $f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta^t$ can be expanded as$f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta = (h_\theta(x_i))^2 - 2h_\theta(x_i) + y_i + y_i^2 + \lambda\theta^t\theta^t$
- then taking the gradient of this expression we can see $\nabla f_i(\theta) = 2x_i^T x_i - 2x_i^t y_i + 2\lambda\theta$
- then we can take the expectation of this as $E[\nabla f_i(\theta)] = E[2x_i^T x_i - 2x_i^t y_i + 2\lambda\theta]$ everything is constant except for x_i, y_i thus $E[\nabla f_i(\theta)] = 2[x_i^t x_i] - 2E[x_i^t y_i] + 2\lambda\theta]$
- then we can see that $2\theta E[x_i^t x^i] = 2\Sigma_{i=1}^{n} x_i^t x_i P(x_i = i)$ and as $P(x_i = i)$ uniformly distributed we can write$2\theta E[x_i^t x_i] = 2\Sigma_{i=1}^{n} x_i^t x_i P(x_i = i) = \frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i$
- similar logic tells us that $[x_i^t y_i] = \frac{1}{m}\Sigma_{i=1}^{m} x_i^t y_i$
- so thus $E[\nabla f_i(\theta)] = 2\theta E[x_i^t x_i] - 2E[x_i^t y_i] + 2\lambda\theta] = 2(\theta\frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i - \frac{1}{m}\Sigma_{i=1}^{m} x_i^t y_i + \lambda h_\theta) = 2\frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i - x_i^t y_i) + 2\lambda\theta$
- we can also see that $\nabla(J_\lambda(\theta)) = \frac{1}{m}\Sigma_{i=1}^{m} 2(x_i)^t x_i - 2x_i^t y_i + 2\lambda\theta$
- thus we see $E[\nabla f_i(\theta)] = \nabla(J_\lambda(\theta))$ which was what we wanted to show

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

- we can write stochastic gradient descent in general as $\theta \leftarrow \theta - \alpha\nabla_\theta \ell(h_\theta x_i, y_i))$
- so in the case of ridge regression it is $\theta \leftarrow \theta - \alpha(2\frac{1}{n} h_\theta x_i^t x_i - x_i^t y_i) + 2\lambda\theta)$

24. Implement `stochastic_grad_descent`.

- ```
  def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2,
  ↪ num_epoch=1000, eta0=False,c=None):
      """
      In this question you will implement stochastic gradient descent
  ↪ with regularization term

      Args:
          X - the feature vector, 2D numpy array of size(num_instances,
  ↪ num_features)
          y - the label vector, 1D numpy array of size(num_instances)
          alpha - string or float, step size in gradient descent
                  NOTE: In SGD, it's not a good idea to use a fixed step
  ↪ size. Usually it's set to 1/sqrt(t) or 1/t
  ```
  ```

21 Q21 (Optional) **1 / 1**

✓ **- 0 pts** *Correct*

- 0.5 pts partially wrong fi

- 1 pts empty

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.

- it is frailly obvious that using $f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta$ we can maintain equality

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

- the expression $f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta^t$ can be expanded as$f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta = (h_\theta(x_i))^2 - 2h_\theta(x_i) + y_i + y_i^2 + \lambda\theta^t\theta^t$
- then taking the gradient of this expression we can see $\nabla f_i(\theta) = 2x_i^T x_i - 2x_i^t y_i + 2\lambda\theta$
- then we can take the expectation of this as $E[\nabla f_i(\theta)] = E[2x_i^T x_i - 2x_i^t y_i + 2\lambda\theta]$ everything is constant except for x_i, y_i thus $E[\nabla f_i(\theta)] = 2[x_i^t x_i] - 2E[x_i^t y_i] + 2\lambda\theta]$
- then we can see that $2\theta E[x_i^t x^i] = 2\Sigma_{i=1}^{n} x_i^t x_i P(x_i = i)$ and as $P(x_i = i)$ uniformly distributed we can write$2\theta E[x_i^t x_i] = 2\Sigma_{i=1}^{n} x_i^t x_i P(x_i = i) = \frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i$
- similar logic tells us that $[x_i^t y_i] = \frac{1}{m}\Sigma_{i=1}^{m} x_i^t y_i$
- so thus $E[\nabla f_i(\theta)] = 2\theta E[x_i^t x_i] - 2E[x_i^t y_i] + 2\lambda\theta] = 2(\theta\frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i - \frac{1}{m}\Sigma_{i=1}^{m} x_i^t y_i + \lambda h_\theta) = 2\frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i - x_i^t y_i) + 2\lambda\theta$
- we can also see that $\nabla(J_\lambda(\theta)) = \frac{1}{m}\Sigma_{i=1}^{m} 2(x_i)^t x_i - 2x_i^t y_i + 2\lambda\theta$
- thus we see $E[\nabla f_i(\theta)] = \nabla(J_\lambda(\theta))$ which was what we wanted to show

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

- we can write stochastic gradient descent in general as $\theta \leftarrow \theta - \alpha\nabla_\theta \ell(h_\theta x_i, y_i))$
- so in the case of ridge regression it is $\theta \leftarrow \theta - \alpha(2\frac{1}{n} h_\theta x_i^t x_i - x_i^t y_i) + 2\lambda\theta)$

24. Implement `stochastic_grad_descent`.

- ```
  def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2,
  ↪ num_epoch=1000, eta0=False,c=None):
      """
      In this question you will implement stochastic gradient descent
  ↪ with regularization term

      Args:
          X - the feature vector, 2D numpy array of size(num_instances,
  ↪ num_features)
          y - the label vector, 1D numpy array of size(num_instances)
          alpha - string or float, step size in gradient descent
                  NOTE: In SGD, it's not a good idea to use a fixed step
  ↪ size. Usually it's set to 1/sqrt(t) or 1/t
  ```
  ```

22 Q22 (Optional) **1 / 1**

✓ **- 0 pts** *Correct*

- 0.5 pts Partially correct

- 1 pts No answer

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.

- it is frailly obvious that using $f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta$ we can maintain equality

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

- the expression $f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta^t$ can be expanded as$f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta = (h_\theta(x_i))^2 - 2h_\theta(x_i) + y_i + y_i^2 + \lambda\theta^t\theta^t$
- then taking the gradient of this expression we can see $\nabla f_i(\theta) = 2x_i^T x_i - 2x_i^t y_i + 2\lambda\theta$
- then we can take the expectation of this as $E[\nabla f_i(\theta)] = E[2x_i^T x_i - 2x_i^t y_i + 2\lambda\theta]$ everything is constant except for x_i, y_i thus $E[\nabla f_i(\theta)] = 2[x_i^t x_i] - 2E[x_i^t y_i] + 2\lambda\theta]$
- then we can see that $2\theta E[x_i^t x^i] = 2\Sigma_{i=1}^{n} x_i^t x_i P(x_i = i)$ and as $P(x_i = i)$ uniformly distributed we can write$2\theta E[x_i^t x_i] = 2\Sigma_{i=1}^{n} x_i^t x_i P(x_i = i) = \frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i$
- similar logic tells us that $[x_i^t y_i] = \frac{1}{m}\Sigma_{i=1}^{m} x_i^t y_i$
- so thus $E[\nabla f_i(\theta)] = 2\theta E[x_i^t x_i] - 2E[x_i^t y_i] + 2\lambda\theta] = 2(\theta\frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i - \frac{1}{m}\Sigma_{i=1}^{m} x_i^t y_i + \lambda h_\theta) = 2\frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i - x_i^t y_i) + 2\lambda\theta$
- we can also see that $\nabla(J_\lambda(\theta)) = \frac{1}{m}\Sigma_{i=1}^{m} 2(x_i)^t x_i - 2x_i^t y_i + 2\lambda\theta$
- thus we see $E[\nabla f_i(\theta)] = \nabla(J_\lambda(\theta))$ which was what we wanted to show

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

- we can write stochastic gradient descent in general as $\theta \leftarrow \theta - \alpha\nabla_\theta \ell(h_\theta x_i, y_i))$
- so in the case of ridge regression it is $\theta \leftarrow \theta - \alpha(2\frac{1}{n} h_\theta x_i^t x_i - x_i^t y_i) + 2\lambda\theta)$

24. Implement `stochastic_grad_descent`.

- ```
  def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2,
  ↪ num_epoch=1000, eta0=False,c=None):
      """
      In this question you will implement stochastic gradient descent
  ↪ with regularization term

      Args:
          X - the feature vector, 2D numpy array of size(num_instances,
  ↪ num_features)
          y - the label vector, 1D numpy array of size(num_instances)
          alpha - string or float, step size in gradient descent
                  NOTE: In SGD, it's not a good idea to use a fixed step
  ↪ size. Usually it's set to 1/sqrt(t) or 1/t
  ```
  ```

```
                if alpha is a float, then the step size in every step
↪ is the float.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t).
                if alpha == "1/t", alpha = 1/t.
        lambda_reg - the regularization coefficient
        num_epoch - number of epochs to go through the whole training
↪ set

    Returns:
        theta_hist - the history of parameter vector, 3D numpy array
↪ of size(num_epoch, num_instances, num_features)
                    for instance, theta in epoch 0 should be
↪ theta_hist[0], theta in epoch(num_epoch) is theta_hist[-1]
        loss hist - the history of loss function vector, 2D numpy
↪ array of size(num_epoch, num_instances)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) # Initialize theta
    theta_hist = np.zeros((num_epoch, num_instances, num_features)) #
    ↪ Initialize t
    loss_hist = np.zeros((num_epoch, num_instances)) # Initialize
    ↪ loss_hist
    random = np.random.default_rng()
    index = np.arange(X.shape[0])
    for i in range(1, num_epoch + 1):
        random.shuffle(index)
        for j, (x, y_) in enumerate(zip(X[index], y[index])):
            if alpha == 'C/sqrt(t)':
                eta = c / np.sqrt(i+1)
            elif alpha=='1/t':
                eta=c/(i+1)
            else:
                eta=alpha
        calc_theta=lambda theta:(x.T * (x.T @ theta - y_) +2 *
        ↪ lambda_reg * theta)
        theta = theta - 2*eta*calc_theta(theta)
        theta_hist[i - 1, j] = theta
        loss_hist[i - 1, j] = compute_square_loss(X, y, theta)
    return theta_hist,loss_hist,
```

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?

23 Q23 (Optional) **0.5 / 1**

- **0 pts** Correct

✓ **- 0.5 pts** *$$\nabla f_i(\theta)$$ not calculated / miscalculated*

- **0.5 pts** Update function incorrect

- **1 pts** No answer

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.

- it is frailly obvious that using $f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta$ we can maintain equality

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

- the expression $f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta^t$ can be expanded as$f_i(\theta) = (h_\theta x_i - y_i)^2 + \lambda\theta^t\theta = (h_\theta(x_i))^2 - 2h_\theta(x_i) + y_i + y_i^2 + \lambda\theta^t\theta^t$
- then taking the gradient of this expression we can see $\nabla f_i(\theta) = 2x_i^T x_i - 2x_i^t y_i + 2\lambda\theta$
- then we can take the expectation of this as $E[\nabla f_i(\theta)] = E[2x_i^T x_i - 2x_i^t y_i + 2\lambda\theta]$ everything is constant except for x_i, y_i thus $E[\nabla f_i(\theta)] = 2[x_i^t x_i] - 2E[x_i^t y_i] + 2\lambda\theta]$
- then we can see that $2\theta E[x_i^t x^i] = 2\Sigma_{i=1}^{n} x_i^t x_i P(x_i = i)$ and as $P(x_i = i)$ uniformly distributed we can write$2\theta E[x_i^t x_i] = 2\Sigma_{i=1}^{n} x_i^t x_i P(x_i = i) = \frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i$
- similar logic tells us that $[x_i^t y_i] = \frac{1}{m}\Sigma_{i=1}^{m} x_i^t y_i$
- so thus $E[\nabla f_i(\theta)] = 2\theta E[x_i^t x_i] - 2E[x_i^t y_i] + 2\lambda\theta] = 2(\theta\frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i - \frac{1}{m}\Sigma_{i=1}^{m} x_i^t y_i + \lambda h_\theta) = 2\frac{1}{n}\Sigma_{i=1}^{m} x_i^t x_i - x_i^t y_i) + 2\lambda\theta$
- we can also see that $\nabla(J_\lambda(\theta)) = \frac{1}{m}\Sigma_{i=1}^{m} 2(x_i)^t x_i - 2x_i^t y_i + 2\lambda\theta$
- thus we see $E[\nabla f_i(\theta)] = \nabla(J_\lambda(\theta))$ which was what we wanted to show

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

- we can write stochastic gradient descent in general as $\theta \leftarrow \theta - \alpha\nabla_\theta \ell(h_\theta x_i, y_i))$
- so in the case of ridge regression it is $\theta \leftarrow \theta - \alpha(2\frac{1}{n} h_\theta x_i^t x_i - x_i^t y_i) + 2\lambda\theta)$

24. Implement `stochastic_grad_descent`.

- ```
  def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2,
  ↪ num_epoch=1000, eta0=False,c=None):
      """
      In this question you will implement stochastic gradient descent
  ↪ with regularization term

      Args:
          X - the feature vector, 2D numpy array of size(num_instances,
  ↪ num_features)
          y - the label vector, 1D numpy array of size(num_instances)
          alpha - string or float, step size in gradient descent
                  NOTE: In SGD, it's not a good idea to use a fixed step
  ↪ size. Usually it's set to 1/sqrt(t) or 1/t
  ```
  ```

```
                if alpha is a float, then the step size in every step
↪ is the float.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t).
                if alpha == "1/t", alpha = 1/t.
        lambda_reg - the regularization coefficient
        num_epoch - number of epochs to go through the whole training
↪ set

    Returns:
        theta_hist - the history of parameter vector, 3D numpy array
↪ of size(num_epoch, num_instances, num_features)
                    for instance, theta in epoch 0 should be
↪ theta_hist[0], theta in epoch(num_epoch) is theta_hist[-1]
        loss hist - the history of loss function vector, 2D numpy
↪ array of size(num_epoch, num_instances)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) # Initialize theta
    theta_hist = np.zeros((num_epoch, num_instances, num_features)) #
    ↪ Initialize t
    loss_hist = np.zeros((num_epoch, num_instances)) # Initialize
    ↪ loss_hist
    random = np.random.default_rng()
    index = np.arange(X.shape[0])
    for i in range(1, num_epoch + 1):
        random.shuffle(index)
        for j, (x, y_) in enumerate(zip(X[index], y[index])):
            if alpha == 'C/sqrt(t)':
                eta = c / np.sqrt(i+1)
            elif alpha=='1/t':
                eta=c/(i+1)
            else:
                eta=alpha
        calc_theta=lambda theta:(x.T * (x.T @ theta - y_) +2 *
        ↪ lambda_reg * theta)
        theta = theta - 2*eta*calc_theta(theta)
        theta_hist[i - 1, j] = theta
        loss_hist[i - 1, j] = compute_square_loss(X, y, theta)
    return theta_hist,loss_hist,
```

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?

24 Q24 (Optional) **1 / 1**

✓ **- 0 pts** *Correct*

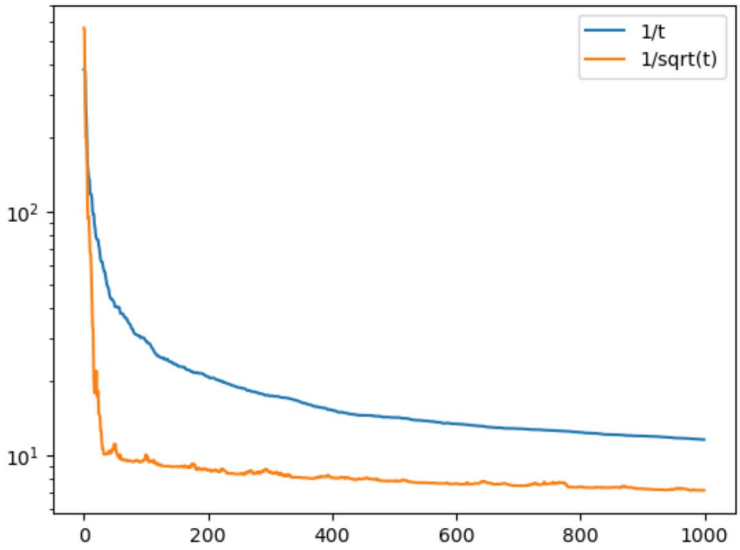
- 0.5 pts Partially correct

- 1 pts No Answer

```
                if alpha is a float, then the step size in every step
↪ is the float.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t).
                if alpha == "1/t", alpha = 1/t.
        lambda_reg - the regularization coefficient
        num_epoch - number of epochs to go through the whole training
↪ set

    Returns:
        theta_hist - the history of parameter vector, 3D numpy array
↪ of size(num_epoch, num_instances, num_features)
                    for instance, theta in epoch 0 should be
↪ theta_hist[0], theta in epoch(num_epoch) is theta_hist[-1]
        loss hist - the history of loss function vector, 2D numpy
↪ array of size(num_epoch, num_instances)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) # Initialize theta
    theta_hist = np.zeros((num_epoch, num_instances, num_features)) #
    ↪ Initialize t
    loss_hist = np.zeros((num_epoch, num_instances)) # Initialize
    ↪ loss_hist
    random = np.random.default_rng()
    index = np.arange(X.shape[0])
    for i in range(1, num_epoch + 1):
        random.shuffle(index)
        for j, (x, y_) in enumerate(zip(X[index], y[index])):
            if alpha == 'C/sqrt(t)':
                eta = c / np.sqrt(i+1)
            elif alpha=='1/t':
                eta=c/(i+1)
            else:
                eta=alpha
        calc_theta=lambda theta:(x.T * (x.T @ theta - y_) +2 *
        ↪ lambda_reg * theta)
        theta = theta - 2*eta*calc_theta(theta)
        theta_hist[i - 1, j] = theta
        loss_hist[i - 1, j] = compute_square_loss(X, y, theta)
    return theta_hist,loss_hist,
```

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?

- it does not seem like fixed α values are able to converge very well for me.
- 

- it seems that using $\alpha = \frac{c}{t}$ the values of θ reaches a point where it converges very quickly but that point does not necessarily get training error as low as it can go
- it seems that using $\alpha = \frac{c}{\sqrt{t}}$ the values of θ reach near convergence a bit more slowly, but result in a lower training loss in the long term

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

25 Q25 (Optional) **1 / 1**

✓ **- 0 pts** *Correct*

- 0.5 pts Missing plots

- 0.5 pts Missing comparison and reasoning

- 0.5 pts Flawed plots / reasoning

- 1 pts No / wrong answer

Dataset

In this problem set we will examine a classification problem. To do so we will use the MNIST dataset[2] 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 784 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 $\boldsymbol{x} \in \mathbb{R}^{784}$, $\boldsymbol{\theta} \in \mathbb{R}^{784}$ 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

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

- recall that logistic loss can be written as $\ell_{log} = log(1 + e^{-(yh_{\theta,b}(\boldsymbol{x}))})$
- we can think of the risk over our dataset $\{\boldsymbol{x}_i, y_i\}_{i=1}^m$ as $\frac{1}{m}\Sigma_{i=1}^m \ell(x_i, y_i)$ given we are using logistic loss this becomes $\frac{1}{m}\Sigma_{i=1}^m log(1 + e^{-(y_i h_{\theta,b}(\boldsymbol{x}_i))})$
- further note that for any given data point $\frac{1}{2}\ell_{log}(x_i, y_i) = (1 + y_i) \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y_i) \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)})$ given $y_i, h_{\theta,b}(\boldsymbol{x}_i) \in \{-1, 1\}$ to show we can check all values that this expression can take
  - suppose $y_i = 1, h_{\theta,b}(\boldsymbol{x}_i) = 1$ then $(1 + y_i) \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y_i) \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)}) = (2) \log(1 + e^{-1}) + (0) \log(1 + e^1) = (2) \log(1 + e^{-1})$ while $\frac{1}{2} log(1 + e^{-(y_i h_{\theta,b}(\boldsymbol{x}_i))}) = \frac{1}{2} log(1 + e^{-1})$
  - suppose $y_i = 1, h_{\theta,b}(\boldsymbol{x}_i) = -1$ then $(1 + y_i) \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y_i) \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)}) = (2) \log(1 + e^1) + (0) \log(1 + e^{-1}) = (2) \log(1 + e^1)$ while $\frac{1}{2} log(1 + e^{-(y_i h_{\theta,b}(\boldsymbol{x}_i))}) = \frac{1}{2} log(1 + e^1)$
  - suppose $y_i = -1, h_{\theta,b}(\boldsymbol{x}_i) = -1$ then $(1 + y_i) \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y_i) \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)}) = (0) \log(1 + e^1) + (2) \log(1 + e^{-1}) = (2) \log(1 + e^{-1})$ while $\frac{1}{2} log(1 + e^{-(y_i h_{\theta,b}(\boldsymbol{x}_i))}) = \frac{1}{2} log(1 + e^{-1})$
  - finally suppose $y_i = -1, h_{\theta,b}(\boldsymbol{x}_i) = 1$ then $(1 + y_i) \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y_i) \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)}) = (0) \log(1 + e^1) + (2) \log(1 + e^1) = (2) \log(1 + e^1)$ while $\frac{1}{2} log(1 + e^{-(y_i h_{\theta,b}(\boldsymbol{x}_i))}) = \frac{1}{2} log(1 + e^1)$

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

- thus we can see that $\frac{1}{2m}(\Sigma_{i=1}^{M}(1+y_i)\log(1+e^{-h_{\theta,b}(\boldsymbol{x}_i)})+(1-y_i)\log(1+e^{h_{\theta,b}(\boldsymbol{x}_i)}) = (0)\log(1+e^1)+(2)\log(1+e^1)$ which was the realtion we wanted to show

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

    - adding an ℓ_1 pelaty with parameter α our loss function becomes $L(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(1+y_i)\log(1+e^{-h_{\theta,b}(\boldsymbol{x}_i)})+(1-y_i)\log(1+e^{h_{\theta,b}(\boldsymbol{x}_i)})+\alpha|\theta|_1$.

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[3], 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)`.

    - ```
      def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2,
      ↪ num_epoch=1000, eta0=False,c=None):
          """
          In this question you will implement stochastic gradient descent
      ↪ with regularization term

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

          Returns:
              theta_hist - the history of parameter vector, 3D numpy array
      ↪ of size(num_epoch, num_instances, num_features)
      ```

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

26 Q26 **1 / 1**

✓ **- 0 pts** *Correct*

- 0.5 pts incomplete proof, need to show how y takes -1 and 1 leads to the final equation

- 1 pts missing proof

- thus we can see that $\frac{1}{2m}(\Sigma_{i=1}^{M}(1+y_i)\log(1+e^{-h_{\theta,b}(\boldsymbol{x}_i)})+(1-y_i)\log(1+e^{h_{\theta,b}(\boldsymbol{x}_i)}) = (0)\log(1+e^1)+(2)\log(1+e^1)$ which was the realtion we wanted to show

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

    - adding an ℓ_1 pelaty with parameter α our loss function becomes $L(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(1+y_i)\log(1+e^{-h_{\theta,b}(\boldsymbol{x}_i)})+(1-y_i)\log(1+e^{h_{\theta,b}(\boldsymbol{x}_i)})+\alpha|\theta|_1$.

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[3], 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)`.

    - ```
      def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2,
      ↪ num_epoch=1000, eta0=False,c=None):
          """
          In this question you will implement stochastic gradient descent
      ↪ with regularization term

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

          Returns:
              theta_hist - the history of parameter vector, 3D numpy array
      ↪ of size(num_epoch, num_instances, num_features)
      ```

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

```
                    for instance, theta in epoch 0 should be
↪   theta_hist[0], theta in epoch(num_epoch) is theta_hist[-1]
        loss hist - the history of loss function vector, 2D numpy
↪   array of size(num_epoch, num_instances)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) # Initialize theta
    theta_hist = np.zeros((num_epoch, num_instances, num_features)) #
    ↪  Initialize t
    loss_hist = np.zeros((num_epoch, num_instances)) # Initialize
    ↪  loss_hist
    random = np.random.default_rng()
    index = np.arange(X.shape[0])
    for i in range(1, num_epoch + 1):
        random.shuffle(index)
        for j, (x, y_) in enumerate(zip(X[index], y[index])):
            if alpha == 'C/sqrt(t)':
                eta = c / np.sqrt(i+1)
            elif alpha=='1/t':
                eta=c/(i+1)
            else:
                eta=alpha
        calc_theta=lambda theta:(x.T * (x.T @ theta - y_) +2 *
        ↪  lambda_reg * theta)
        theta = theta - 2*eta*calc_theta(theta)
        theta_hist[i - 1, j] = theta
        loss_hist[i - 1, j] = compute_square_loss(X, y, theta)
    return theta_hist,loss_hist,
```

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.

27 Q27 **1 / 1**

✓ **- 0 pts** *Correct*

- 1 pts incorrect, should be adding $$\alpha||\theta||_1$$ to the previous loss function

- thus we can see that $\frac{1}{2m}(\Sigma_{i=1}^{M}(1+y_i)\log(1+e^{-h_{\theta,b}(\boldsymbol{x}_i)})+(1-y_i)\log(1+e^{h_{\theta,b}(\boldsymbol{x}_i)}) = (0)\log(1+e^1)+(2)\log(1+e^1)$ which was the realtion we wanted to show

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

    - adding an ℓ_1 pelaty with parameter α our loss function becomes $L(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(1+y_i)\log(1+e^{-h_{\theta,b}(\boldsymbol{x}_i)})+(1-y_i)\log(1+e^{h_{\theta,b}(\boldsymbol{x}_i)})+\alpha|\theta|_1$.

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[3], 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)`.

    - ```
      def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2,
      ↪ num_epoch=1000, eta0=False,c=None):
          """
          In this question you will implement stochastic gradient descent
      ↪ with regularization term

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

          Returns:
              theta_hist - the history of parameter vector, 3D numpy array
      ↪ of size(num_epoch, num_instances, num_features)
      ```

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

```
                    for instance, theta in epoch 0 should be
↪   theta_hist[0], theta in epoch(num_epoch) is theta_hist[-1]
        loss hist - the history of loss function vector, 2D numpy
↪   array of size(num_epoch, num_instances)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) # Initialize theta
    theta_hist = np.zeros((num_epoch, num_instances, num_features)) #
    ↪  Initialize t
    loss_hist = np.zeros((num_epoch, num_instances)) # Initialize
    ↪  loss_hist
    random = np.random.default_rng()
    index = np.arange(X.shape[0])
    for i in range(1, num_epoch + 1):
        random.shuffle(index)
        for j, (x, y_) in enumerate(zip(X[index], y[index])):
            if alpha == 'C/sqrt(t)':
                eta = c / np.sqrt(i+1)
            elif alpha=='1/t':
                eta=c/(i+1)
            else:
                eta=alpha
        calc_theta=lambda theta:(x.T * (x.T @ theta - y_) +2 *
        ↪  lambda_reg * theta)
        theta = theta - 2*eta*calc_theta(theta)
        theta_hist[i - 1, j] = theta
        loss_hist[i - 1, j] = compute_square_loss(X, y, theta)
    return theta_hist,loss_hist,
```

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.

28 Q28 **0 / 1**

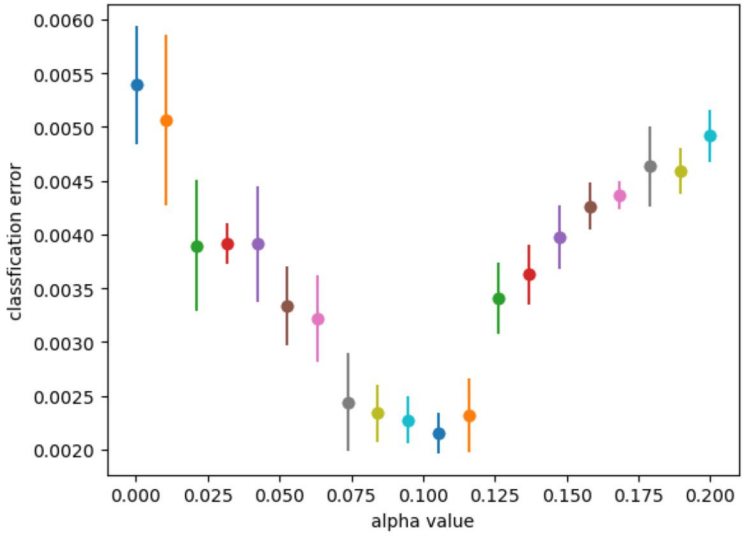
- **0 pts** Correct
- **0.5 pts** Partially correct

✓ **- 1 pts** *Wrong / no answer*

```
                    for instance, theta in epoch 0 should be
↪   theta_hist[0], theta in epoch(num_epoch) is theta_hist[-1]
        loss hist - the history of loss function vector, 2D numpy
↪   array of size(num_epoch, num_instances)
    """
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) # Initialize theta
    theta_hist = np.zeros((num_epoch, num_instances, num_features)) #
    ↪  Initialize t
    loss_hist = np.zeros((num_epoch, num_instances)) # Initialize
    ↪  loss_hist
    random = np.random.default_rng()
    index = np.arange(X.shape[0])
    for i in range(1, num_epoch + 1):
        random.shuffle(index)
        for j, (x, y_) in enumerate(zip(X[index], y[index])):
            if alpha == 'C/sqrt(t)':
                eta = c / np.sqrt(i+1)
            elif alpha=='1/t':
                eta=c/(i+1)
            else:
                eta=alpha
        calc_theta=lambda theta:(x.T * (x.T @ theta - y_) +2 *
        ↪  lambda_reg * theta)
        theta = theta - 2*eta*calc_theta(theta)
        theta_hist[i - 1, j] = theta
        loss_hist[i - 1, j] = compute_square_loss(X, y, theta)
    return theta_hist,loss_hist,
```

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.



-
- i chose to widen the the search range of alpha a bit, since i think it makes the overall trend more clear
- namely that reducing α only causes lower average and standard deviation in classification error to a point, after which these metric increase again

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

    - our estimator is based on stochastic gradient descent, repeating the experiment 10 times, at each alpha level helps reduce sampling uncertainty and thus approximation error

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

    - it looks like .01 is the optimal value of α

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 784 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?

29 Q29 **2 / 2**

✓ **- 0 pts** *Correct.*

- 0.5 pts x-axis not in log scale.

- 0.5 pts Fewer than 10 alpha values in the specified range.

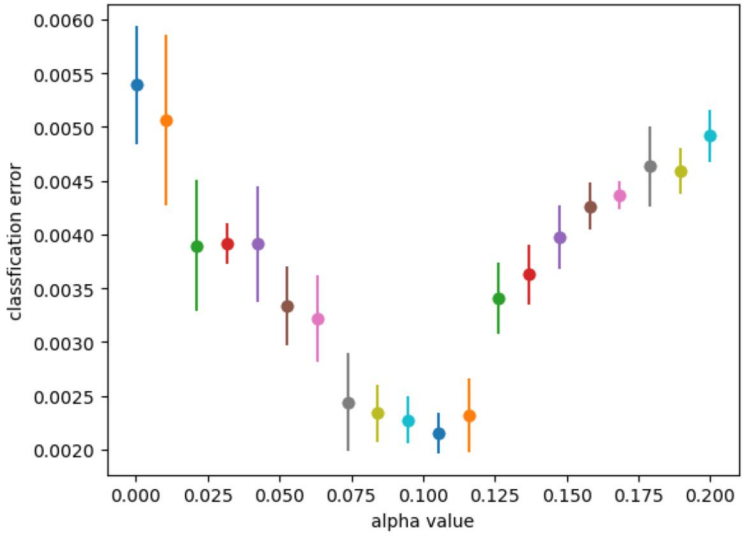
- 0.5 pts Missing some error bars.

- 1 pts Should use L1 regularization, not L2.

- 1 pts No error bars or wrong standard errors

- 1.5 pts No or wrong classification errors.

- 2 pts No answer shown.



-
- i chose to widen the the search range of alpha a bit, since i think it makes the overall trend more clear
- namely that reducing α only causes lower average and standard deviation in classification error to a point, after which these metric increase again

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

    - our estimator is based on stochastic gradient descent, repeating the experiment 10 times, at each alpha level helps reduce sampling uncertainty and thus approximation error

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

    - it looks like .01 is the optimal value of α

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 784 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?

30 Q30 **0.5 / 1**

- 0 pts Correct

✓ **- 0.5 pts** *The answer is too vague or partial. Sources of randomness should come from*

*1). the **random initialization** of the parameters θ and b*

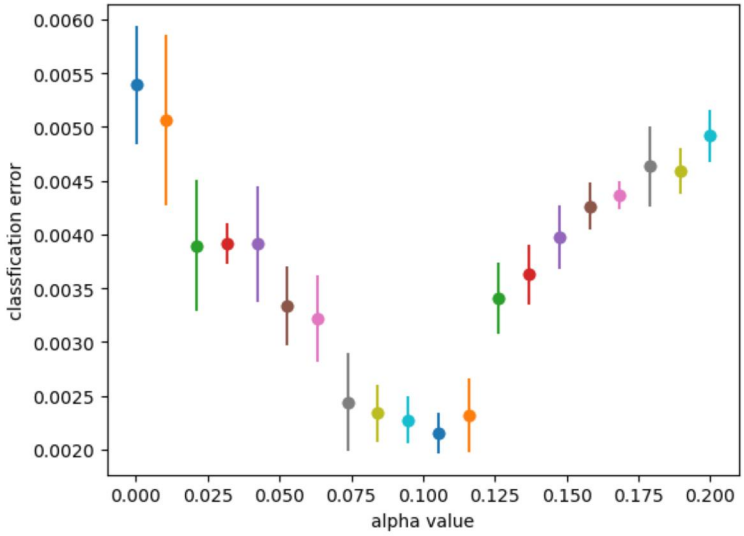
*2). the **random shuffling** of the same 100 samples in SGD in each experiment*

*Both will cause the **gradient path** to differ every round.*

(Note we do not subsample data each time. Need to be clear about why SGDClassifier selects different paths each time by mentioning random shuffling/order.)

- 1 pts Wrong.

- 1 pts No answer shown.



-
- i chose to widen the the search range of alpha a bit, since i think it makes the overall trend more clear
- namely that reducing α only causes lower average and standard deviation in classification error to a point, after which these metric increase again

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

    - our estimator is based on stochastic gradient descent, repeating the experiment 10 times, at each alpha level helps reduce sampling uncertainty and thus approximation error

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

    - it looks like .01 is the optimal value of α

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 784 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?

31 Q31 **0.5 / 1**

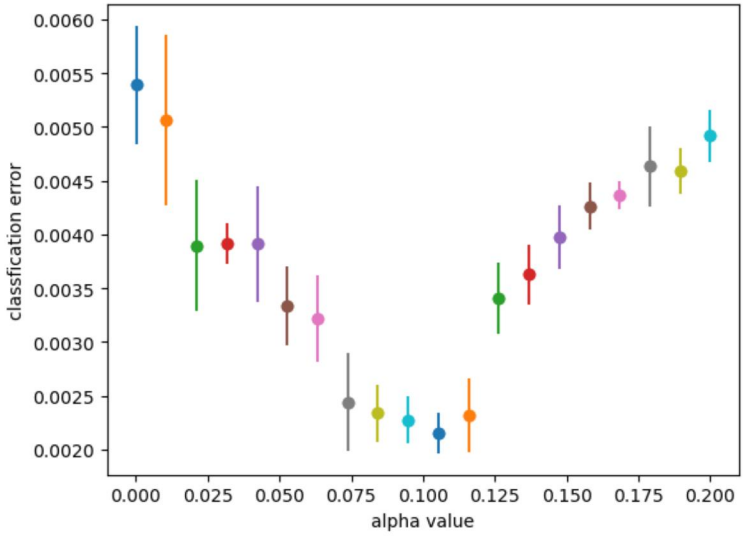
- 0 pts Correct alpha with the minimum classification error.

✓ **- 0.5 pts** *The optimal value of alpha does not match the one with the lowest error in the plot, or no plot as reference.*

- 0.5 pts Should choose only one optimal value.

- 0.5 pts The trend of the plot itself is not correct.

- 1 pts No answer was shown.



-
- i chose to widen the the search range of alpha a bit, since i think it makes the overall trend more clear
- namely that reducing α only causes lower average and standard deviation in classification error to a point, after which these metric increase again

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

    - our estimator is based on stochastic gradient descent, repeating the experiment 10 times, at each alpha level helps reduce sampling uncertainty and thus approximation error

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

    - it looks like .01 is the optimal value of α

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 784 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?

32 Q32 **0.5 / 1**

- 0 pts Correct

✓ **- 0.5 pts** *Didn't display the 10 plots corresponding to 10 different values of alpha.*

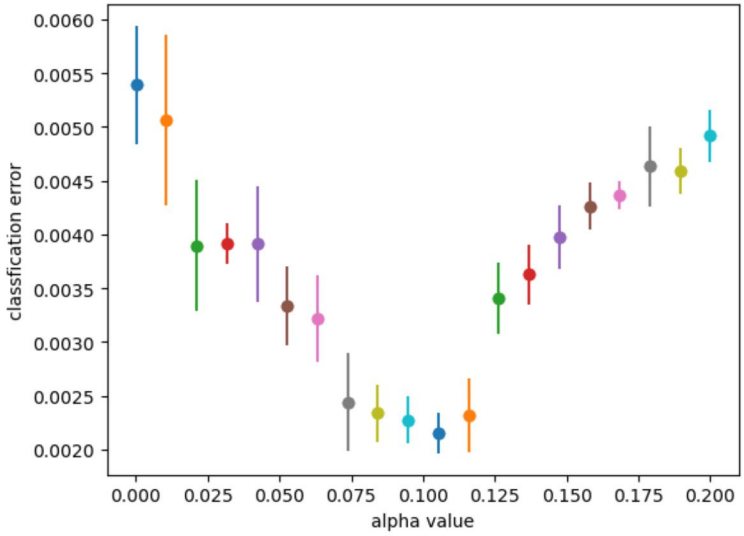
- 0.5 pts Wrong plot type. Should use plt.imshow() to show the plot in 2-d dimension.

- 0.5 pts Theta trend is not correct or not clear. Should be more sparse as alpha increases.

- 0.5 pts Wrong coef values or scales.

- Coef should be reshaped to 28x28 array.
- Use one appropriate scale for all plots to show trends better.

- 1 pts No answer shown.



-
- i chose to widen the the search range of alpha a bit, since i think it makes the overall trend more clear
- namely that reducing α only causes lower average and standard deviation in classification error to a point, after which these metric increase again

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

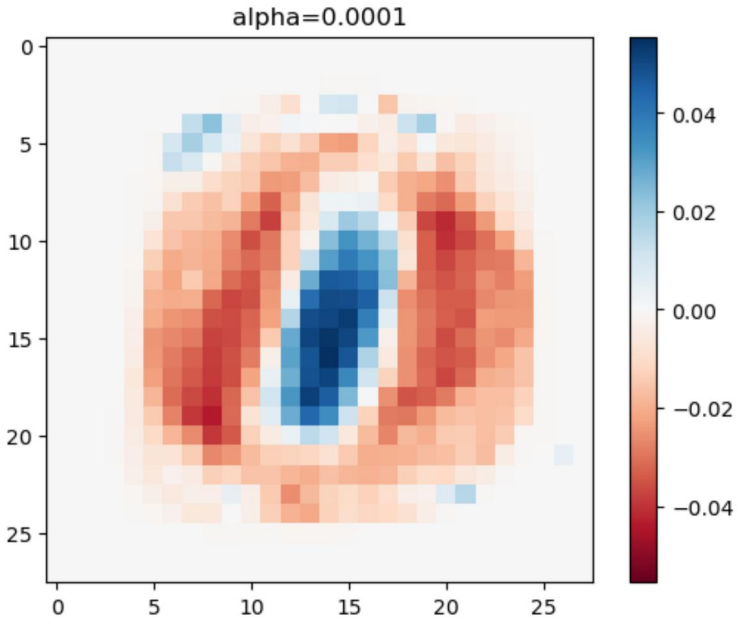
    - our estimator is based on stochastic gradient descent, repeating the experiment 10 times, at each alpha level helps reduce sampling uncertainty and thus approximation error

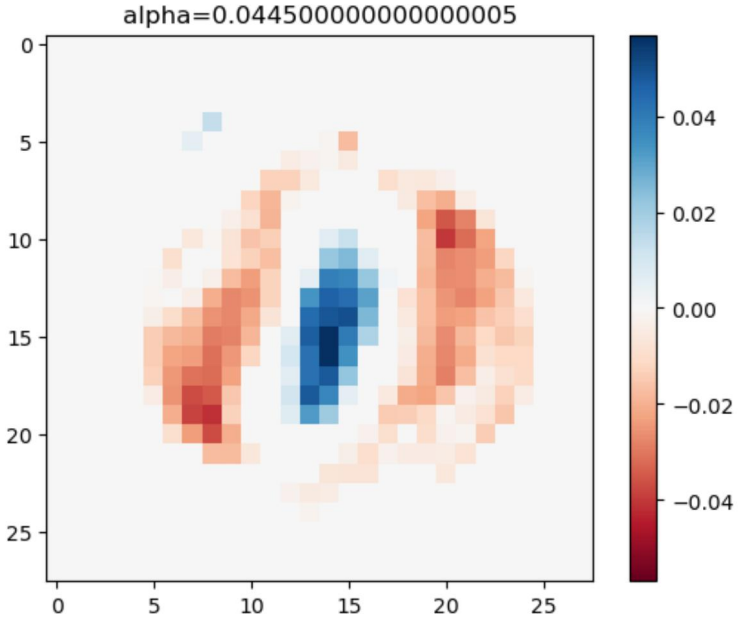
31. What is the optimal value of the parameter α among the values you tested?

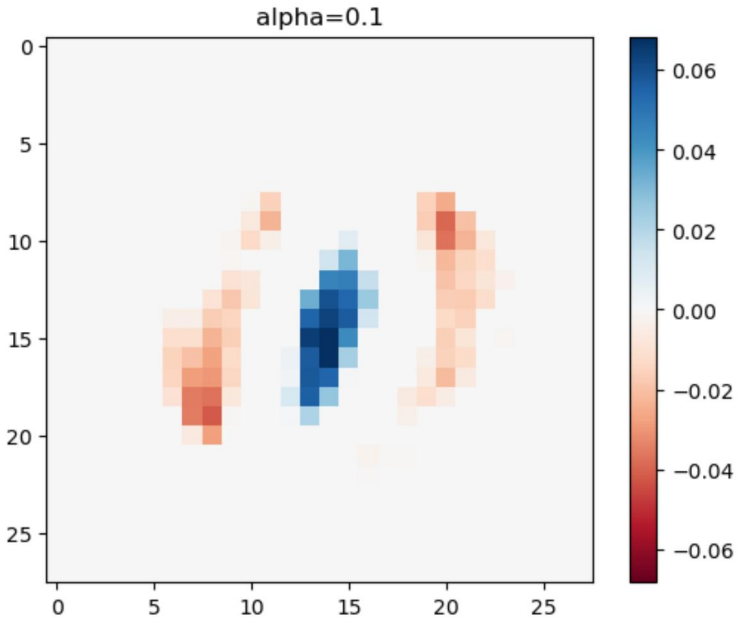
    - it looks like .01 is the optimal value of α

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 784 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?

- 
alpha=0.0001
0
5
10
15
20
25
0.04
0.02
0.00
−0.02
−0.04

- 
alpha=0.044500000000000005
0
5
10
15
20
25
0.04
0.02
0.00
−0.02
−0.04



- as can be seen from this subset of the images over the range higher values of α tend to have less color implying that more of the coefficients are close to zero which makes sense because we are using ℓ_1 regularization with a stronger regularization term and thus will get a sparse estimator

33 Q33 **0.5 / 1**

- 0 pts Correct

✓ **- 0.5 pts** *Didn't mention or wrong about the pattern recognized by* θ. *(detecting the shape of positive and negative values from the image). Only talked about the effect of regularization.*

- 0.5 pts Didn't mention or wrong about the effect of regularization. (making the coefficients more sparse.)

- 1 pts Didn't mention the pattern of θ nor the effect of regularization on sparsity.

- 1 pts No answer shown.