Homework 2: Gradient Descent & Regularization

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

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

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

1 Statistical Learning Theory

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

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

1. Show that:
$$\text{Training Error} = \frac{1}{N}\left|\left|\Big(X(X^TX)^{-1}X^T - I\Big)\epsilon\right|\right|_2^2$$
where $\epsilon \sim \mathcal{N}(0, \sigma^2 I_n)$ and training error is defined as $\frac{1}{N}||X\hat{b} - y||_2^2$.

We want to show that:
$$\frac{1}{N}\left|\left|\Big(X(X^TX)^{-1}X^T - I\Big)\epsilon\right|\right|_2^2 = \frac{1}{N}||X\hat{b} - y||_2^2$$
The best way to do that is to simply show that:
$$\begin{aligned}
\Big(X(X^TX)^{-1}X^T - I\Big)\epsilon &= X\hat{b} - y \\
X(X^TX)^{-1}X^T\epsilon - \epsilon I &= X(X^TX)^{-1}X^T y - y \\
X(X^TX)^{-1}X^T\epsilon - \epsilon &= X(X^TX)^{-1}X^T(Xb + \epsilon) - (Xb + \epsilon) \\
X(X^TX)^{-1}X^T\epsilon - \epsilon &= X(X^TX)^{-1}X^T Xb + X(X^TX)^{-1}X^T\epsilon - Xb - \epsilon \\
X(X^TX)^{-1}X^T\epsilon - \epsilon &= Xb + X(X^TX)^{-1}X^T\epsilon - Xb - \epsilon \\
X(X^TX)^{-1}X^T\epsilon - \epsilon &= X(X^TX)^{-1}X^T\epsilon - \epsilon
\end{aligned}$$
et violà!

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

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?

Given $A = X(X^TX)^{-1}X^T$, we can show that:

$$\begin{aligned} A^T &= \left(X(X^TX)^{-1}X^T\right)^T \\ &= X\left((X^TX)^{-1}\right)^T X^T \\ &= X(X^TX)^{-1}X^T = A \end{aligned}$$

Since $A^T = A$, it means that A is symmetric and that $A^2 = A^TA$. In addition, we can also show that $A^2 = A$:

$$\begin{aligned} A^2 &= \left(X(X^TX)^{-1}X^T\right)\left(X(X^TX)^{-1}X^T\right) \\ &= X(X^TX)^{-1}X^TX(X^TX)^{-1}X^T \\ &= X(X^TX)^{-1}X^T = A \end{aligned}$$

By definition, the eigenvalues of A and A^2 are:

$$\begin{aligned} Av &= \lambda v \\ \Rightarrow A^2v &= \lambda Av \\ \Rightarrow A^2v &= \lambda^2 v \\ \Rightarrow \lambda v &= \lambda^2 v \\ \Rightarrow \lambda &= \lambda^2 \Rightarrow \lambda = 1 \text{ or } 0 \end{aligned}$$

Next, by definition, the trace of a matrix is equal to the sum of its eigenvalues. In addition, a special property of trace is that, given a matrix B and a matrix C, $\text{Tr}(BC) = \text{Tr}(CB)$. Since X is a $N \times d$ matrix:

$$\text{Tr}(A) = \text{Tr}(X(X^TX)^{-1}X^T) = \text{Tr}(X^TX(X^TX)^{-1}) = \text{Tr}(I_d) = d$$

Now, back to the expectation of the training error, we can plug in A:

$$
\begin{aligned}
&\mathbb{E}\left[\frac{1}{N}\left|\left|\Big(X(X^TX)^{-1}X^T - I\Big)\epsilon\right|\right|_2^2\right] \\
&= \frac{1}{N}\,\mathbb{E}\left[\left|\left|(A-I)\epsilon\right|\right|_2^2\right] \\
&= \frac{1}{N}\,\mathbb{E}\left[\Big(\epsilon^T(A-I)^T(A-I)\epsilon\Big)\right] \\
&= \frac{1}{N}\,\mathbb{E}\left[\epsilon^T(A-I)^2\epsilon\right] \\
&= \frac{1}{N}\,\mathbb{E}\left[\epsilon^T(I-A)\epsilon\right] \\
&= \frac{1}{N}\Big[\mathbb{E}\left(\epsilon^T\epsilon\right) - \mathbb{E}\left(\epsilon^T A\epsilon\right)\Big] \\
&= \frac{1}{N}\left[\sum_{i=1}^{N}{\epsilon_i}^2 - \sum_{i=1}^{N}\sum_{j=1}^{N}\epsilon_i A_{i,j}\epsilon_j\right]
\end{aligned}
$$

For the summation in the former half in the bracket, ${\epsilon_i}^2$ is given at σ^2. For the double summation in the latter half in the bracket, when $i \neq j$, $A_{i,j}$ is just a constant, while ϵ_i and ϵ_j are independent, so the expectation is zero. When $i = j$, the summation of $A_{i,j}$ is simply the trace of A, while ϵ_i and ϵ_j are equal, so the expectation is σ^2. We have already shown that the trace of A is d. Therefore, we have:

$$
\begin{aligned}
&\frac{1}{N}\left[\sum_{i=1}^{N}{\epsilon_i}^2 - \sum_{i=1}^{N}\sum_{j=1}^{N}\epsilon_i A_{i,j}\epsilon_j\right] \\
&= \frac{1}{N}\Big(N\sigma^2 - d\sigma^2\Big) \\
&= \frac{N-d}{N}\sigma^2
\end{aligned}
$$

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

   When d is close to N, the nominator of the fraction above becomes really small. Therefore, the expectation of the training error is very low when we overfit the data.

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.

```python
def feature_normalization(train, test):

    train_normalized = np.zeros(train.shape)
    test_normalized  = np.zeros(test.shape)
    for i in range(len(train[0])):
        col = train[:,i]
        tmax = np.max(col)
        tmin = np.min(col)
        if tmax == tmin:
            train_normalized[:,i] = 2
        else:
            train_normalized[:,i] = (col - tmin) / (tmax - tmin)
            test_normalized[:,i] = (test[:,i] - tmin) / (tmax - tmin)

    for i in range(len(train_normalized[0])):
        tmin = np.min(train_normalized[:,i])
        if tmin > 1:
            np.delete(train_normalized, i, 1)
            np.delete(test_normalized, i, 1)

    return train_normalized, test_normalized
```

   Yes, NumPy's broadcasting is used here.

At the end of the skeleton code, the function `load_data` loads, split into a training and test set, and normalize the data using your `feature_normalization`.

Linear regression

In linear regression, we consider the hypothesis space of linear functions $h_\theta : \mathbb{R}^d \to \mathbb{R}$, where

$$h_\theta(x) = \theta^T x,$$

for $\theta, \boldsymbol{x} \in \mathbb{R}^d$, and we choose θ that minimizes the following "average square loss" objective function:

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

where $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m) \in \mathbb{R}^d \times \mathbb{R}$ is our training data.
While this formulation of linear regression is very convenient, it's more standard to use a hypothesis space of affine functions:

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

which allows a nonzero intercept term b – sometimes called a "bias" term. The standard way to achieve this, while still maintaining the convenience of the first representation, is to add an extra dimension to $\boldsymbol{x}$ that is always a fixed value, such as 1, and use $\theta, x \in \mathbb{R}^{d+1}$. Convince yourself that this is equivalent. We will assume this representation.

5. Let $X \in \mathbb{R}^{m \times (d+1)}$ be the *design matrix*, where the i'th row of X is $\boldsymbol{x}_i$. Let $y = (y_1, \ldots, y_m)^T \in \mathbb{R}^{m \times 1}$ be the *response*. Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign. [1]

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} (h_\theta(\boldsymbol{x}_i) - y_i)^2 \\ &= \frac{1}{m} \sum_{i=1}^{m} (\theta^\top \boldsymbol{x}_i - y_i)^2 \\ &= \frac{1}{m} \|X\theta - y\|^2 \end{aligned}$$

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

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

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

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla J(\theta_t)$$

[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.

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):

    m = len(X)
    f = (X @ theta) - y
    loss = np.dot(f,f) / m

    return loss
```

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):

    m = len(X)
    f = X @ theta - y
    grad = (2 * np.transpose(X) @ f) / m

    return grad
```

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.

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

    true_gradient = compute_square_loss_gradient(X, y, theta)
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features)

    h = np.identity(num_features)
    J_plus = compute_square_loss(X, y, (theta + epsilon @ h))
    J_minus = compute_square_loss(X, y, (theta - epsilon @ h))
    approx_grad = J_plus - J_minus / (2 * epsilon)

    diff = true_gradient - approx_grad
    dist = np.sqrt(np.dot(diff.T, diff))
    if dist > tolerance:
        answer = False
    else:
        answer = True

    return answer
```

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.

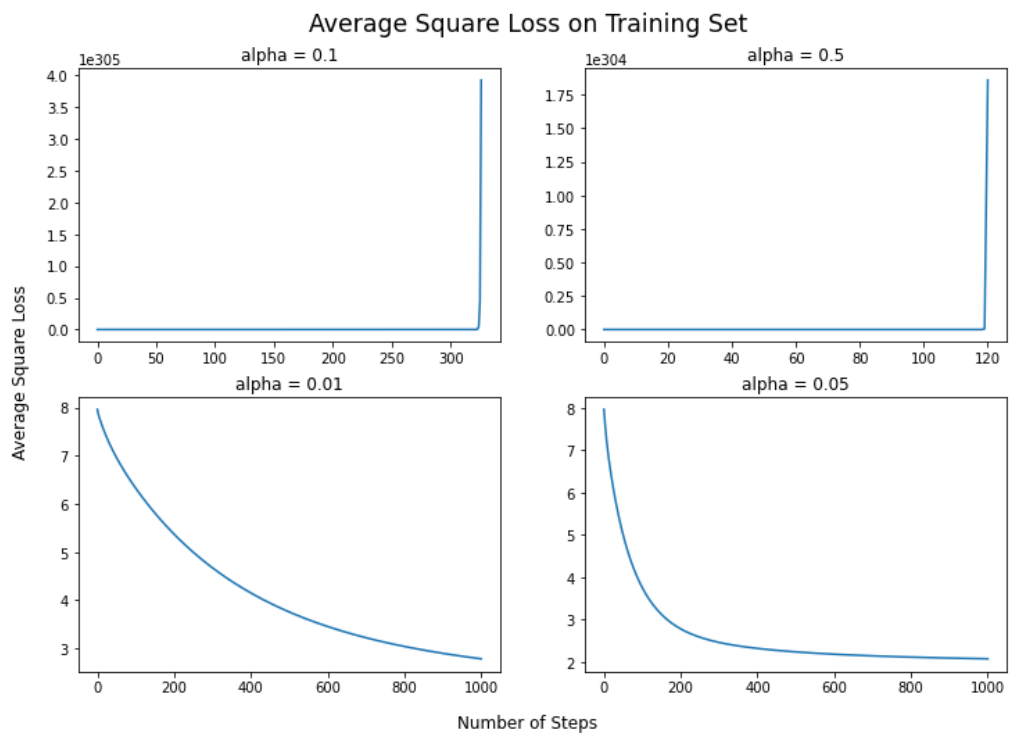
11. Complete `batch_gradient_descent`. Note the phrase *batch* gradient descent distinguishes between *stochastic* gradient descent or more generally *minibatch* gradient descent.

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

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

    for n in range(num_step+1):
        theta_hist[n] = theta
        loss_hist[n] = compute_square_loss(X, y, theta)
        grad = compute_square_loss_gradient(X, y, theta)
        if grad_check == True:
            assert grad_checker
        theta = theta - alpha * grad

    return theta_hist, loss_hist
```

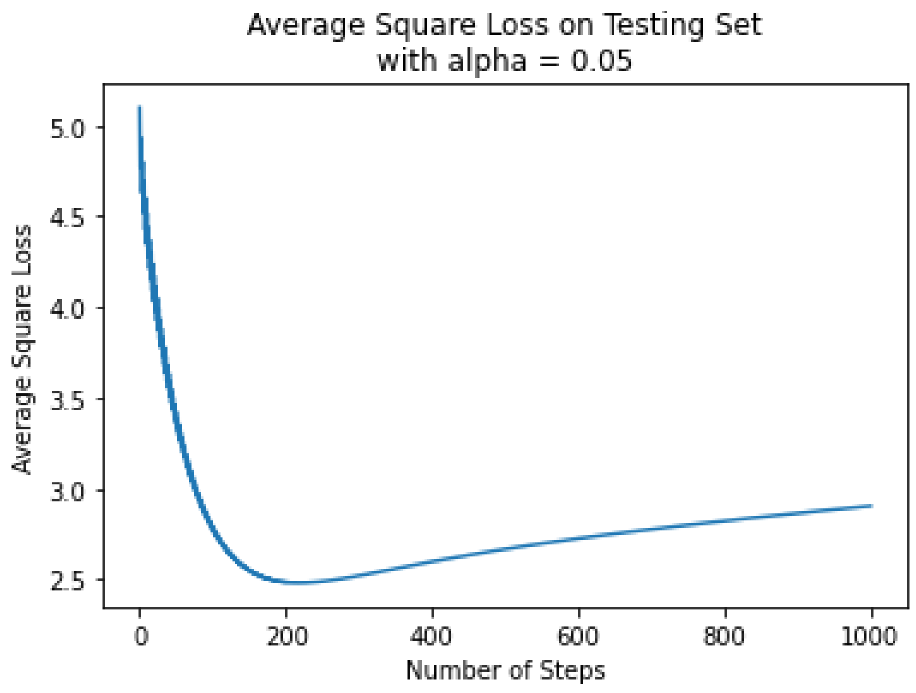
12. Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge. Starting with a step-size of 0.1, try various different fixed step sizes to see which converges most quickly and/or which diverge. As a minimum, try step sizes 0.5, 0.1, .05, and .01.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

    For the four different step sizes that were tested, we can see that the average square loss on the training set actually diverges as the number of steps increases when the step size is either 0.1 or 0.5. On the other hand, when the step size is either 0.01 or 0.05, we observe convergence as the number of steps increases. The average square loss converges most quickly with $\alpha = 0.05$.

    (Graphs on next page)



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



Ridge Regression

We will add ℓ_2 regularization to linear regression. When we have a large number of features compared to instances, regularization can help control overfitting. *Ridge regression* is linear regression with ℓ_2 regularization. The regularization term is sometimes called a penalty term. The objective function for ridge regression is

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

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)

    From question 6, we already have the gradient of $J(\theta)$. So, now, we only have to compute the gradient for $\lambda\theta^\top\theta$:

$$\lambda\theta^\top\theta = \lambda\|\theta\|^2 \;\Rightarrow\; \nabla\lambda\|\theta\|^2 = 2\lambda\theta$$

    which gives us the gradient of $J_\lambda(\theta)$ and the expression for updating θ as:

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

15. Implement compute_regularized_square_loss_gradient.

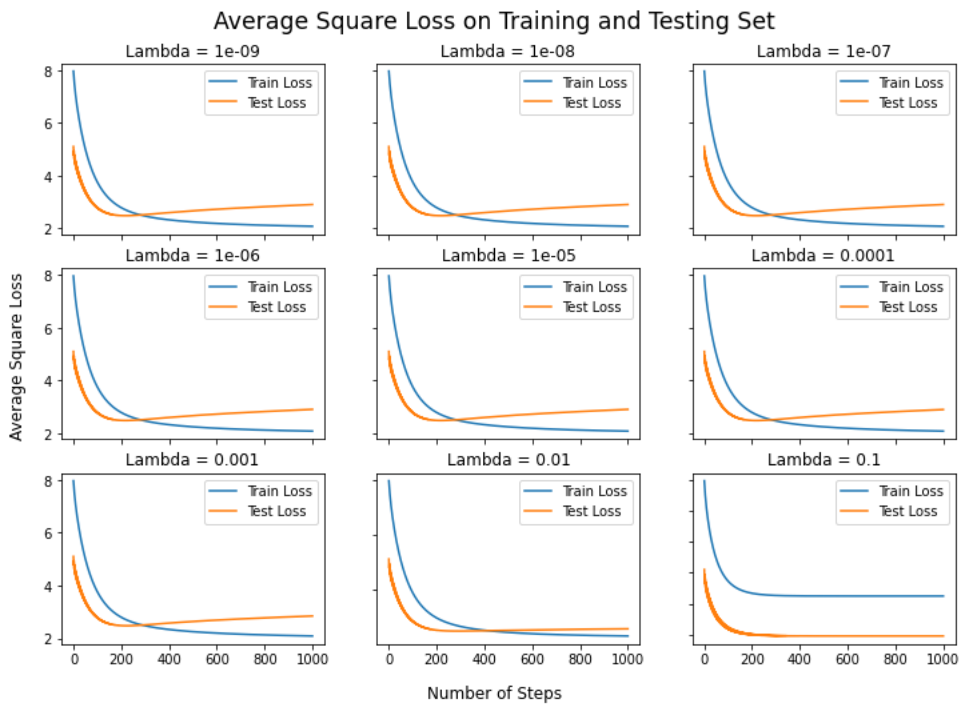
```
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):

    m = len(X)
    f = X @ theta - y
    ridge = 2 * lambda_reg * theta
    grad = ((2 * np.transpose(X) @ f) / m) + ridge

    return grad
```

16. Implement regularized_grad_descent.

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

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

    for n in range(num_step+1):
        theta_hist[n] = theta
        loss_hist[n] = compute_square_loss(X, y, theta)
        grad = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)
        theta = theta - alpha * grad

    return theta_hist, loss_hist
```

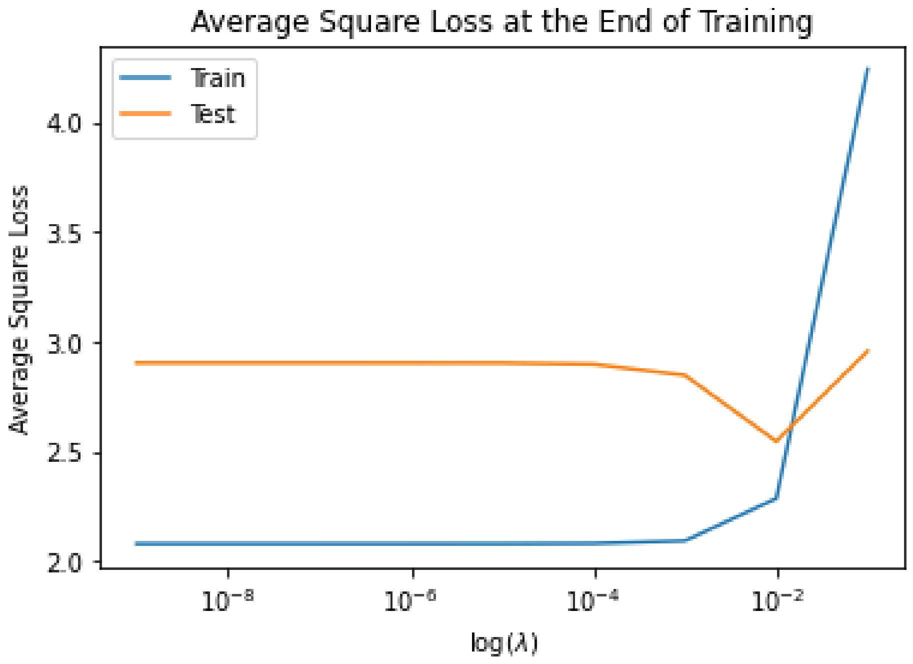
Our goal is to find λ that gives the minimum average square loss on the test set. So you should start your search very broadly, looking over several orders of magnitude. For example, $\lambda \in \left\{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\right\}$. Then you can zoom in on the best range. Follow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

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



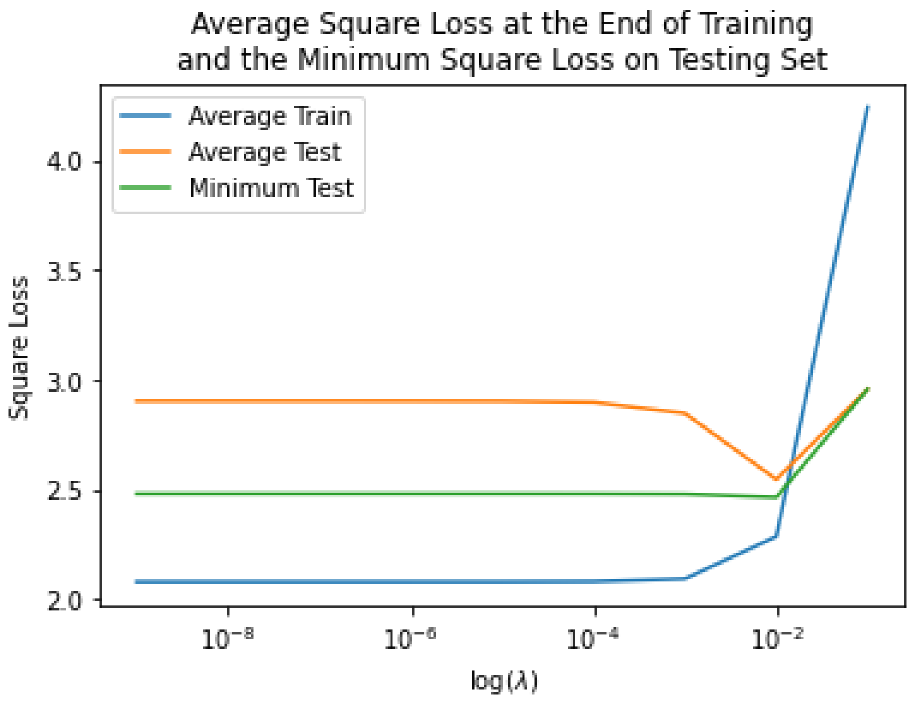
Based on the nine different λ values tested, we observe overfitting as the number of steps increases for all of them except for $\lambda = 0.1$. In addition, we can see that as λ increases, the overfitting seems to begin later (most obvious when $\lambda = 0.01$).

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 ?



The λ I would chosse is 0.001.

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 λ value you would select with early stopping the same as before?

    Yes, the λ value I would select with early stopping is the same as before.

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

    In practice, I would select the λ where both the testing square loss and training square loss are low and similar. If the training square loss is higher than the testing square loss, the model is likely overfitting. If the testing square loss is a lot higher than the training error, then it likely means the testing set is not large enough. Therefore, in this case, the λ I would select is still 0.001.

3 Image classification with regularized logistic regression

Dataset

In this second problem set we will examine a classification problem. To do so we will use the MNIST dataset[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

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

The Log Loss function is given as below if we plug in the margin m:

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

for each sample (x_i, y_i). For $y_i = -1$, we have:

$$\ell_{logistic} = \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)})$$

and for $y_i = 1$, we have:

$$\ell_{logistic} = \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)})$$

That means for all y_i, the Log Loss function is:

$$\ell_{logistic} = (1 + y_i) \cdot \log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y_i) \cdot \log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)})$$

When $y_i = -1$, the former half is zero, and when $y_i = 1$, the latter half is zero. Therefore, the objective function can be given as below:

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

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

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

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

We are going to use the module `SGDClassifier` from scikit learn. In the code provided we have set a little example of its usage. By checking the online documentation[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.

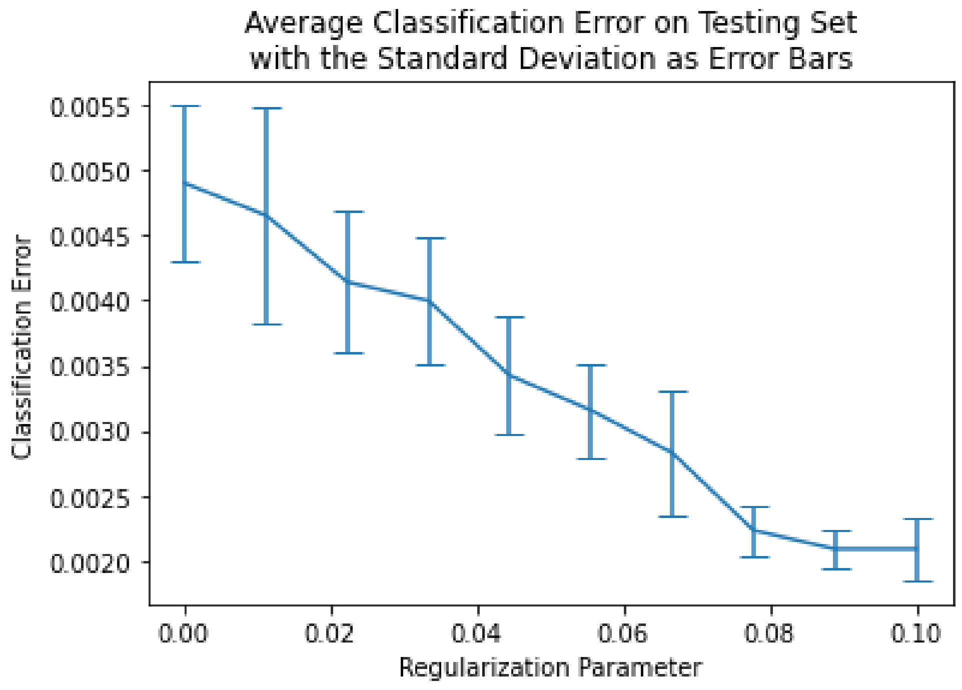
28. To evaluate the quality of our model we will use the classification error, which corresponds to the fraction of incorrectly labeled examples. For a given sample, the classification error is 1 if no example was labeled correctly and 0 if all examples were perfectly labeled. Using the method `clf.predict()` 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 classification_error(clf, X, y):

    y_hat = clf.predict(X)

    count = 0
    for i in range(len(y)):
        if y[i] != y_hat[i]:
            count += 1

    return count / len(y)
```

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

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

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



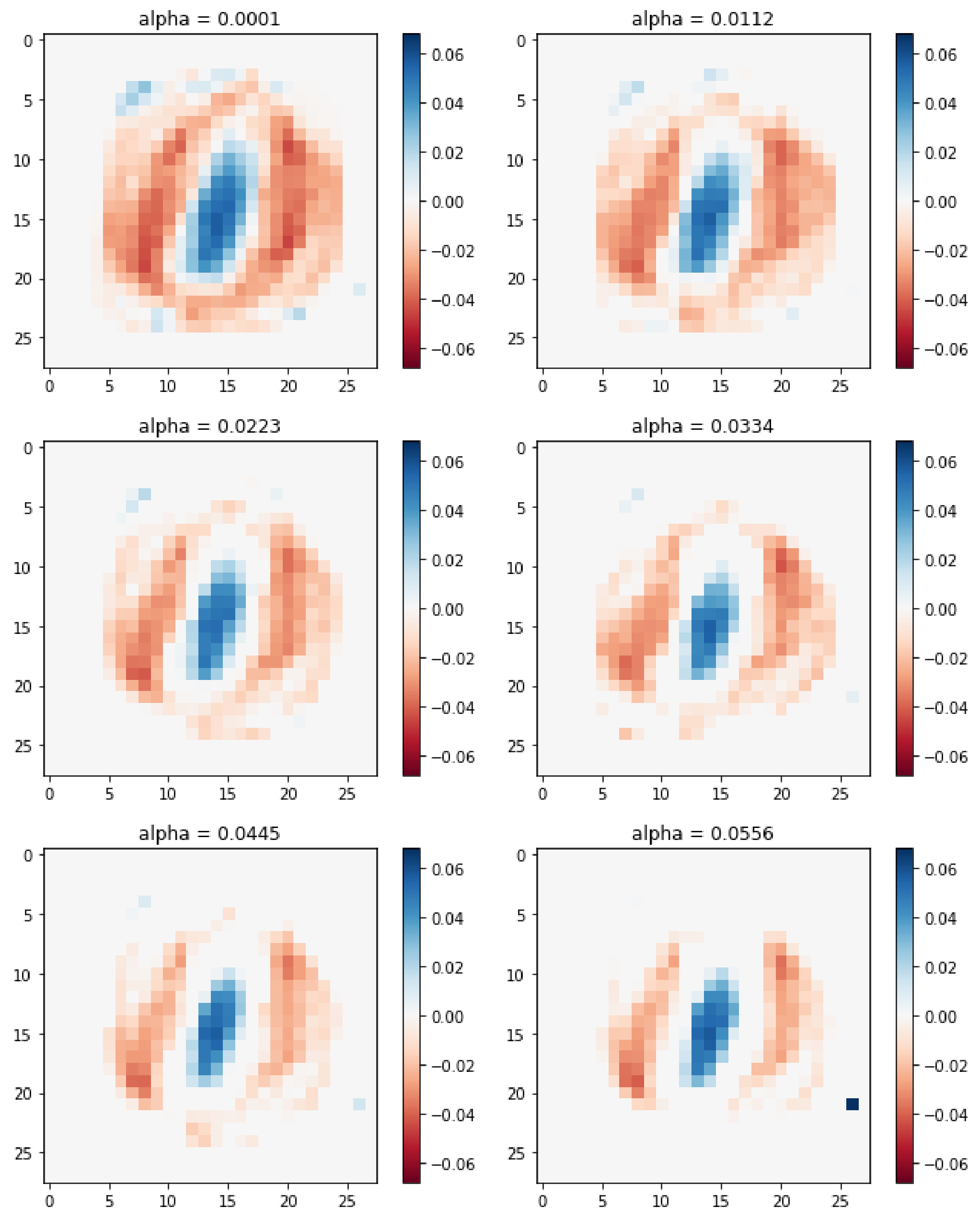
30. Which source(s) of randomness are we averaging over by repeating the experiment?

    By repeating the experiment, we are averaging over the randomness created by the sub-sampling process. The original data set had more than nine thousand rows for both the training set and the testing set, but in this experiment we are limiting both sets to one hundred rows.

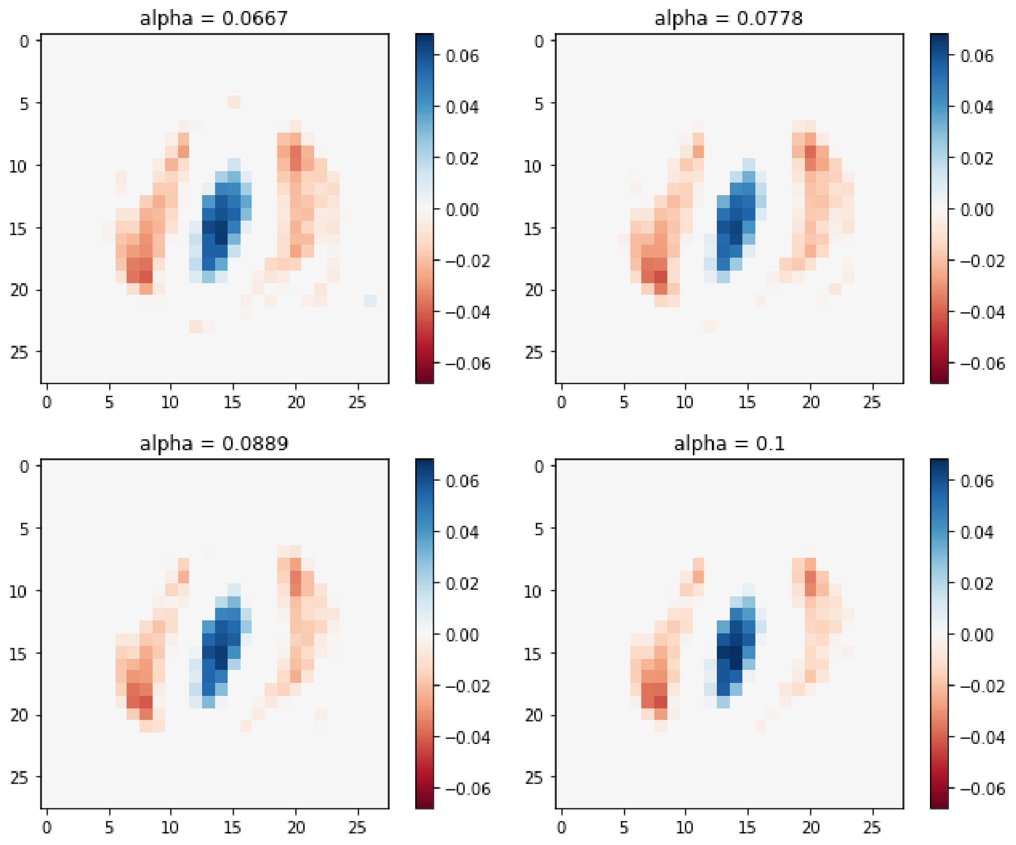
31. What is the optimal value of the parameter α among the values you tested?

    The average classification error is the lowest (and, perhaps coincidentally, the standard deviation is the smallest) at $\alpha = 0.0889$; so, that is the optimal value of the parameter α among the ten values I tested.

32. Finally, for one run of the fit for each value of α plot the value of the fitted θ. You can access it via `clf.coef_`, and should reshape the 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.



(continues on next page)



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

    As α increases, we observe more and more grey boxes, which means that more and more of the θ approach zero. This makes sense because the effect of regularization is to reduce the number of features used, so having zero (or a value close to zero) as the coefficient essentially eliminates the given feature from the model.

Find codes at: `https://github.com/anchengyoung/MS_Data_Science/blob/main/machine_learning/hw2/homework_2.py`