

Homework 2: Gradient Descent & Regularization

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

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

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

1 Statistical Learning Theory

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

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

1. Show that:

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

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

Let $A = X(X^T X)^{-1} X^T$, and $y = Xb + \epsilon$ (note that this is b and not \hat{b}). Also note the following property:

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

Which shows that A is symmetric with eigenvalues strictly equal to 1 or 0.

To show the equivalency we begin by doing the following manipulations:

$$\begin{aligned}
\frac{1}{N} \left\| \left(X(X^T X)^{-1} X^T - I \right) \epsilon \right\|_2^2 &= \frac{1}{N} \|X\hat{b} - y\|_2^2 \\
&= \frac{1}{N} \|Ay - y\|_2^2 \\
&= \frac{1}{N} \|(A - I)y\|_2^2 \\
&= \frac{1}{N} \|(A - I)(Xb + \epsilon)\|_2^2 \\
&= \frac{1}{N} \|(A - I)(A + \epsilon)\|_2^2 \\
&= \frac{1}{N} \|A^2 - A + A\epsilon - I\epsilon\|_2^2 \\
&= \frac{1}{N} \|A\epsilon - I\epsilon\|_2^2 \\
&= \frac{1}{N} \|(A - I)\epsilon\|_2^2 \\
&= \frac{1}{N} \left\| \left(X(X^T X)^{-1} X^T - I \right) \epsilon \right\|_2^2
\end{aligned} \tag{2}$$

We firstly substituted our definition for A , then expanded the definition of the Euclidean norm via dot product, foiled the inside terms and manipulated the expression accordingly, resulting in the desired equivalency.

2. Show that the expectation of the training error can be expressed solely in terms of **only** N, d, σ as:

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

Hints:

- Consider $A = X(X^T X)^{-1} X^T$. What is $A^T A$? 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 ?

We have shown that:

$$\begin{aligned}
\text{Training Error} &= \frac{1}{N} \left\| (A - I)\epsilon \right\|_2^2 \\
&= \frac{1}{N} \langle (A - I), (A - I) \rangle \\
&= \frac{1}{N} (A^2 - 2A + I) * \epsilon^2 \\
&= \frac{1}{N} \epsilon^T * (I_N - A) * \epsilon
\end{aligned} \tag{3}$$

Therefore we can take the expectation and manipulate as such:

$$\begin{aligned}
E(\text{Training Error}) &= \frac{1}{N} E(\epsilon^2 * I_N) - E(\epsilon^T * A * \epsilon) \\
&= \frac{1}{N} [E(\sum_{i=1}^n \epsilon^2) - E(\epsilon^T * A * \epsilon)] \\
&= \frac{1}{N} [(\sum_{i=1}^n E(\epsilon^2)) - E(\epsilon^T * A * \epsilon)] \\
&= \frac{1}{N} [n\sigma^2 - E(\epsilon^T * A * \epsilon)] \\
&= \frac{1}{N} [n\sigma^2 - (\sum_{i=1}^d E(\epsilon^2) - (\sum_{i=d+1}^n E(0)))] \\
&= \frac{(N-d)\sigma^2}{N}
\end{aligned} \tag{4}$$

For this exercise, we begun by applying the second hint that $A^2 = A$, then we utilized the expectation to find the expected training error. Then, using linearity of expectations, we can take the sum of expectations. Finally, using the fact that A is square symmetric, and due to the Spectral Theorem can be expressed as $A = PDP^T$, we utilize the fact that the eigenvalues, can be only 1 or 0, and since the rank of A is equal to d , then the eigenmatrix, D will have 1s in the diagonal from i equals 1 to d , and eigenvalues of 0 from $d+1$ to n diagonals. Using the properties of the quadratic form, we know that the expression $\epsilon^T A \epsilon$ is the same as $Tr(A) \times \epsilon^2$.

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.

As we increase the number of features we include in our model, we are essentially increasing the amount of terms in our prediction function. With more terms, we can fit the noise present in our ground truth function better. This overfitting process reduces training error (and without a doubt would increase test error).

In other words, as d approaches N the numerator in the equation we proved in part 2 ($\frac{(N-d)\sigma}{N}$) approaches 0. When that occurs the total training error becomes 0.

2 Gradient descent for ridge(less) linear regression

Dataset

We have provided you with a file called `ridge_regression_dataset.csv`. Columns `x0` through `x47` correspond to the input and column `y` corresponds to the output. We are trying to fit the data using a linear model and gradient based methods. Please also check the supporting code in `skeleton_code.py`. Throughout this problem, we refer to particular blocks of code to help you step by step.

Feature normalization When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization, features with larger values are treated as “more important”, which is not usually desired.

One common approach to feature normalization is perform an affine transformation (i.e. shift and rescale) on each feature so that all feature values in the training set are in $[0, 1]$. Each feature gets its own transformation. We then apply the same transformations to each feature on the validation set or test set. Importantly, the transformation is “learned” on the training set, and then applied to the test set. It is possible that some transformed test set values will lie outside the $[0, 1]$ interval.

4. Modify function `feature_normalization` to normalize all the features to $[0, 1]$. Can you use numpy’s broadcasting here? Often broadcasting can help to simplify and/or speed up your code. Note that a feature with constant value cannot be normalized in this way. Your function should discard features that are constant in the training set.

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

Linear regression

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

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

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

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

where $(x_1, y_1), \dots, (x_m, y_m) \in \mathbb{R}^d \times \mathbb{R}$ 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:

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

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 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 x_i . Let $y = (y_1, \dots, y_m)^T \in \mathbb{R}^{m \times 1}$ be the *response*. Write the objective function $J(\theta)$ as a ma-

trix/vector expression, without using an explicit summation sign.¹

This is a relatively easy task, as our design matrix X can be multiplied by a vector of weights, θ to calculate our predicted \hat{y} values, where $\hat{y} \in \mathbb{R}^{m \times 1}$. Therefore, our loss function can be re-expressed as the following simple linear algebra expression:

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

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

The gradient of our loss function is defined as the vector of partial derivatives with respect to θ . It can be expressed as follows:

$$\begin{aligned} \nabla J(\theta) &= \frac{\partial J}{\partial \theta} \frac{1}{m} \|X\theta - y\|_2^2 \\ &= \frac{\partial J}{\partial \theta} \left(\frac{1}{m} (\theta^T X^T X \theta + y^T y - 2\theta^T X^T y) \right) \\ &= \frac{2}{m} (X^T X \theta - X^T y) \end{aligned} \tag{5}$$

Therefore:

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

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

Our algorithm to update θ is rather simple. To calculate the next theta, iteration $i + 1$, we take the θ from iteration i and subtract the gradient that we calculated above, scaled for some step size η :

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

8. Modify the function `compute_square_loss`, to compute $J(\theta)$ for a given θ . You might want to create a small dataset for which you can compute $J(\theta)$ by hand, and verify that your `compute_square_loss` function returns the correct value.
9. Modify the function `compute_square_loss_gradient`, to compute $\nabla_{\theta} J(\theta)$. You may again want to use a small dataset to verify that your `compute_square_loss_gradient` function returns the correct value.

Gradient checker

We can numerically check the gradient calculation. If $J : \mathbb{R}^d \rightarrow \mathbb{R}$ 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

$$\lim_{\epsilon \rightarrow 0} \frac{J(\theta + \epsilon h) - J(\theta - \epsilon h)}{2\epsilon}.$$

¹Being 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.

It is also given by the more standard definition of directional derivative,

$$\lim_{\epsilon \rightarrow 0} \frac{1}{\epsilon} [J(\theta + \epsilon h) - J(\theta)] .$$

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 $h = (1, 0, 0, \dots, 0)$ to get the first component of the gradient. Then take $h = (0, 1, 0, \dots, 0)$ to get the second component, and so on.

10. Complete the function `grad_checker` according to the documentation of the function given in the `skeleton_code.py`. Alternatively, you may complete the function `generic_grad_checker` so which can work for any objective function.

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

Batch gradient descent

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

11. Complete `batch_gradient_descent`. Note the phrase *batch* gradient descent distinguishes between *stochastic* gradient descent or more generally *minibatch* gradient descent.
12. Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge. Starting with a step-size of 0.1, try various different fixed step sizes to see which converges most quickly and/or which diverge. As a minimum, try step sizes 0.5, 0.1, .05, and .01. Plot the average square loss on the training set as a function of the number of steps for each step size. Briefly summarize your findings.
13. For the learning rate you selected above, plot the average test loss as a function of the iterations. You should observe overfitting: the test error first decreases and then increases.

Ridge Regression

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

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

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)

Gradient of computing $J_\lambda(\theta)$: To begin, let's derive the gradient of ridge regression using linear algebra, similarly with how we did to problem 6.

$$\begin{aligned} J_\lambda(\theta) &= \frac{1}{m} \|X\theta - y\|_2^2 + \lambda \theta^T \theta \\ \nabla J_\lambda(\theta) &= \frac{\partial J}{\partial \theta} \left(\frac{1}{m} (\theta^T X^T X \theta + y^T y - 2\theta^T X^T y) + \lambda \theta^T \theta \right) \\ \nabla J_\lambda(\theta) &= \frac{2}{m} (X^T X \theta - X^T y) + 2\lambda \theta \end{aligned} \tag{6}$$

Updating the gradient descent algorithm with our new gradient:

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

15. Implement `compute_regularized_square_loss_gradient`.
16. Implement `regularized_grad_descent`.

Our goal is to find λ that gives the minimum average square loss on the test set. So you should start your search very broadly, looking over several orders of magnitude. For example, $\lambda \in \{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\}$. 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?
18. Plot the training average square loss and the test average square loss at the end of training as a function of λ . You may want to have $\log(\lambda)$ on the x -axis rather than λ . Which value of λ would you choose?
19. Another heuristic of regularization is to *early-stop* the training when the test error reaches a minimum. Add to the last plot the minimum of the test average square loss along training as a function of λ . Is the value λ you would select with early stopping the same as before?

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

In practice, it is best to select the θ that minimizes testing error the most out of all θ considered. The reason for this is that we want our model to generalize the best to real world data, and our testing split is the best indicator during an experiment of how the model might perform once deployed. In our example, it seems that the value of 10^{-2} minimized the testing error the most, so that is the one we would choose to use.

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 $\{1, \dots, m\}$. 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 i th example. In practical implementations for ML, the data points are randomly shuffled, and then we sweep through the whole training set one by one, and perform an update for each training example individually. One pass through the data is called an *epoch*. Note that each epoch of SGD touches as much data as a single step of batch gradient descent. You can use the same ordering for each epoch, though optionally you could investigate whether reshuffling after each epoch affects the convergence speed.

21. Show that the objective function

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

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.

This can be simply shown by substituting our definition of our loss function, and calculating its risk. Our Ridge Regression Least squares loss function for a single point, (x_i, y_i) is defined as:

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

Taking a batch of size m (m denoting the number of data points used in SGD) we have:

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

And thus, showing the equivalency.

22. Show that the stochastic gradient $\nabla_{\theta} f_i(\theta)$, for i chosen uniformly at random from $\{1, \dots, m\}$, is an *unbiased estimator* of $\nabla_{\theta} J_{\lambda}(\theta)$. In other words, show that $\mathbb{E}[\nabla f_i(\theta)] = \nabla J_{\lambda}(\theta)$ 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 $\mathbb{E}[\nabla f_i(\theta)]$

Firstly, we start with the definition of the Ridge Regression gradient:

$$\nabla f_i(\theta) = \frac{2}{m} (X_i^T X_i \theta - X_i^T y_i) + 2\lambda \theta$$

We can show that the SGD is an unbiased estimator of the full batch gradient by first illustrating what happens when we plug in only 1 data point into our SGD gradient calculator. Note that the probability of selecting any one individual data point is $P(x_i = i) = \frac{1}{m}$.

$$\nabla f_i(\theta) = 2(X_i^T X_i \theta - X_i^T y_i) + 2\lambda \theta$$

We can then apply the expected value to this function. Adding in the probability of selecting any one of our m data points for each m datapoint we can show the equivalency:

$$\begin{aligned} \mathbb{E}[\nabla f_i(\theta)] &= E[2(X_i^T X_i \theta - X_i^T y_i) + 2\lambda \theta] \\ &= 2 \times \frac{1}{m} \times \sum_{i=1}^m (X_i^T X_i \theta - X_i^T y_i) * P(X_i = i) + 2\lambda \theta \\ &= \frac{2}{m} (X^T X \theta - X^T y) + 2\lambda \theta \end{aligned} \quad (7)$$

This proves the equivalency as:

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

Which we accomplished by noting that the probability of selecting any one data point m is:

$$2(X_i^T X_i \theta - X_i^T y_i) * P(X_i = i)$$

After all that we have:

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

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

We simply rewrite the way we would normally update the theta during ridge regression, but instead of using the whole design matrix X and labels, y , we use the subset selected for SGD: the j^{th} row of X and the j^{th} observation in our target vector, y . For each iteration i of our stochastic gradient descent, we update θ as follows:

$$\theta^{i+1} = \theta^i - \eta \nabla f_j(\theta^i) = \theta^i - \eta \left(\frac{2}{m} (X_j^T X_j \theta^i - X_j^T y_j) + 2\lambda \theta^i \right)$$

24. Implement `stochastic_grad_descent`.
25. Use SGD to find θ_λ^* that minimizes the ridge regression objective for the λ you selected in the previous problem. (If you could not solve the previous problem, choose $\lambda = 10^{-2}$). Try a few fixed step sizes (at least try $\eta_t \in \{0.05, .005\}$). Note that SGD may not converge with fixed step size. Simply note your results. Next try step sizes that decrease with the step number according to the following schedules: $\eta_t = \frac{C}{t}$ and $\eta_t = \frac{C}{\sqrt{t}}$, $C \leq 1$. Please include $C = 0.1$ in your submissions. You are encouraged to try different values of C (see notes below for details). For each step size rule, plot the value of the objective function (or the log of the objective function if that is more clear) as a function of epoch (or step number, if you prefer). How do the results compare?

A few remarks about the question above:

- In this case we are investigating the convergence rate of the optimization algorithm with different step size schedules, thus we're interested in the value of the objective function, which includes the regularization term.

- Sometimes the initial step size (C for C/t and C/\sqrt{t}) is too aggressive and will get you into a part of parameter space from which you can't recover. Try reducing C to counter this problem.
- SGD convergence is much slower than GD once we get close to the minimizer (remember, the SGD step directions are very noisy versions of the GD step direction). If you look at the objective function values on a logarithmic scale, it may look like SGD will never find objective values that are as low as GD gets. In statistical learning theory terminology, GD has much smaller *optimization* error than SGD. However, this difference in optimization error is usually dominated by other sources of error (estimation error and approximation error). Moreover, for very large datasets, SGD (or minibatch GD) is much faster (by wall-clock time) than GD to reach a point close enough to the minimizer.

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

3 Image classification with regularized logistic regression

Dataset

In this second problem set we will examine a classification problem. To do so we will use the MNIST dataset² which is one of the traditional image benchmark for machine learning algorithms. We will only load the data from the 0 and 1 class, and try to predict the class from the image. You will find the support code for this problem in `mnist_classification_source_code.py`. Before starting, take a little time to inspect the data. Load `X_train`, `y_train`, `X_test`, `y_test` with `pre_process_mnist_01()`. Using the function `plt.imshow` from `matplotlib` visualize some data points from `X_train` by reshaping the 764 dimensional vectors into 28×28 arrays. Note how the class labels '0' and '1' have been encoded in `y_train`. No need to report these steps in your submission.

Logistic regression

We will use here again a linear model, meaning that we will fit an affine function,

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

with $\mathbf{x} \in \mathbb{R}^{764}$, $\theta \in \mathbb{R}^{764}$ and $b \in \mathbb{R}$. This time we will use the logistic loss instead of the squared loss. Instead of coding everything from scratch, we will also use the package `scikit learn` and study the effects of ℓ_1 regularization. You may want to check that you have a version of the package up to date (0.24.1).

26. Recall the definition of the logistic loss between target y and a prediction $h_{\theta,b}(\mathbf{x})$ as a function of the margin $m = y h_{\theta,b}(\mathbf{x})$. Show that given that we chose the convention $y_i \in \{-1, 1\}$, our objective function over the training data $\{\mathbf{x}_i, y_i\}_{i=1}^m$ can be re-written as

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

We know from Lecture 2 that the definition of the logistic log function is as follows:

$$L(\theta) = \frac{1}{n} \sum_i^n \ln(1 + e^{-m})$$

where n is the number of data points and $m = y_i h_{\theta,b}(x_i)$. Using our definition above, and substituting in our margin for m we have the following statement:

$$L(\theta) = \frac{1}{m} \sum_i^m \ln(1 + e^{-y_i h_{\theta,b}(x_i)}) \text{ where } y \in -1, 1$$

Please note: now the variable m denotes the number of datapoints, and we no longer use the indicator variable n .

We can prove the equivalency by doing a few manipulations. Firstly, lets multiply our statement by $\frac{2}{2}$ to get the following:

$$L(\theta) = \frac{1}{2m} \sum_i^m 2 * \ln(1 + e^{-y_i h_{\theta,b}(x_i)})$$

²<http://yann.lecun.com/exdb/mnist/>

Now lets explore what happens to our equation when y_i takes it's two possible values, (when $y_i \in [-1, 1]$).

Firstly, lets try substituting $y_i = -1$ to illustrate the equivalency we need to prove

$$\begin{aligned}
 L(\theta) &= \frac{1}{2m} \sum_i^m (1 + -1) * \ln(1 + e^{h_{\theta,b}(x_i)}) + (1 + 1) * \ln(1 + e^{h_{\theta,b}(x_i)}) \\
 &= \frac{1}{2m} \sum_i^m (0) * \ln(1 + e^{h_{\theta,b}(x_i)}) + 2 * \ln(1 + e^{h_{\theta,b}(x_i)}) \\
 &= \frac{1}{2m} \sum_i^m 2 * \ln(1 + e^{-h_{\theta,b}(x_i)}) \\
 &= \frac{1}{m} \sum_i^m \ln(1 + e^{-m})
 \end{aligned} \tag{8}$$

After doing the above manipulations, we arrive at the original logistic loss function, thus shoving the equivalency between the two when $y = -1$

Now for our other case, when $y_i = 1$:

$$\begin{aligned}
 L(\theta) &= \frac{1}{2m} \sum_i^m (1 + 1) * \ln(1 + e^{-h_{\theta,b}(x_i)}) + (1 - 1) * \ln(1 + e^{h_{\theta,b}(x_i)}) \\
 &= \frac{1}{2m} \sum_i^m 2 * \ln(1 + e^{-h_{\theta,b}(x_i)}) + 0 * \ln(1 + e^{h_{\theta,b}(x_i)}) \\
 &= \frac{1}{2m} \sum_i^m 2 * \ln(1 + e^{-h_{\theta,b}(x_i)}) \\
 &= \frac{1}{m} \sum_i^m \ln(1 + e^{-m})
 \end{aligned} \tag{9}$$

And again, we arrive at our original logistic loss function, thus showing equivalency between the two functions when $y = 1$. Since the only two cases we can have are $y = 1, y = -1$, we have proven the equivalency between the two functions.

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

If we regularize the loss function similarly to how we have done in previos problems like Ridge Regression, our Logistic Loss Function would change accordingly:

$$L(\theta) = \frac{1}{n} \sum_i^n \ln(1 + e^{-m}) + \alpha \|\theta\|_1$$

Where $\|\theta\|_1$ is the ℓ_1 norm of θ with regularization parameter α .

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 documentation³, make sure you

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

understand the meaning of all the keyword arguments that were specified. We will keep the learning rate schedule and the maximum number of iterations fixed to the given values for all the problem. Note that scikit learn is actually implementing a fancy version of SGD to deal with the ℓ_1 penalty which is not differentiable everywhere, but we will not enter these details here.

28. To evaluate the quality of our model we will use the classification error, which corresponds to the fraction of incorrectly labeled examples. For a given sample, the classification error is 1 if no example was labeled correctly and 0 if all examples were perfectly labeled. Using the method `clf.predict()` from the classifier write a function that takes as input an `SGDClassifier` which we will call `clf`, a design matrix `X` and a target vector `y` and returns the classification error. You should check that your function returns the same value as `1 - clf.score(X, y)`.

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

29. Report the test classification error achieved by the logistic regression as a function of the regularization parameters α (taking 10 values between 10^{-4} and 10^{-1}). You should make a plot with α as the x-axis in log scale. For each value of α , you should repeat the experiment 10 times so has to finally report the mean value and the standard deviation. You should use `plt.errorbar` to plot the standard deviation as error bars.
30. Which source(s) of randomness are we averaging over by repeating the experiment?
31. What is the optimal value of the parameter α among the values you tested?
32. Finally, for one run of the fit for each value of α plot the value of the fitted θ . You can access it via `clf.coef_`, and should reshape the 764 dimensional vector to a 28×28 array 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?