1 Logistic Regression

In this exercise we focus on the classification problem. This is just like the regression problem, except for the fact that the values y (we want to predict) take only a small number of discrete values. We will focus on the **binary** classification problem in which y can take only two values, 0 and 1. For instance, if we are trying to build an email spam classifier, then $\mathbf{x}$ may be the features of the textual description of an email, and y may be 1 if it is a spam mail, and 0 otherwise. The value 0 is also called the negative class, whereas the value of 1 represents the positive class. Additionally, they are sometimes denoted by the symbols '–' and '+'. Given a training example $\mathbf{x}^{(i)}$, the corresponding $y^{(i)}$ is also called the **label**.

1.1 Theory

We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given $\underline{x}$. However, it is easy to construct examples where this method performs poor for the classification case. Intuitively, it also does not make sense for $h(\underline{x})$ to take values larger than 1 or smaller than 0 when we know that $y \in [0, 1]$. To fix this, we modify the form of our hypothesis function $h(\underline{x})$. Thus,

$$h(\underline{x}) = g\left(\underline{\theta}^T \underline{x}\right) = \frac{1}{1 + e^{-\underline{\theta}^T \underline{x}}} \tag{1}$$

where

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

is called the **logistic** (or the **sigmoid**) function. Notice that $g(z)$ tends towards 1 as $z \to \infty$, and $g(z)$ tends towards 0 as $z \to -\infty$. Moreover, $g(z)$, and hence also $h(\underline{x})$, is always bounded between 0 and 1. As before, we are keeping the convention of letting $x_0 = 1$, so that $\underline{\theta}^T \underline{x} = \theta_0 + \sum_{j=1}^{m} \theta_j x_j$. So, given the logistic regression model, how do we fit $\underline{\theta}$ for it? We will use maximum likelihood estimation for that. Let us assume that $P(y = 1 | \underline{x}; \underline{\theta}) = h(\underline{x})$ and $P(y = 0 | \underline{x}; \underline{\theta}) = 1 - h(\underline{x})$. Note that this can be written more compactly as

$$P(y | \underline{x}; \underline{\theta}) = [h(\underline{x})]^y \times [1 - h(\underline{x})]^{1-y}.$$

Assuming that we have n training examples, we can then write down the likelihood of the parameters as

$$\begin{aligned}\mathcal{L}(\underline{\theta}) &= P(\underline{y}|\mathbf{X};\underline{\theta}) \\ &= \prod_{i=1}^{n} P\left(y_i|\underline{x}_i;\underline{\theta}\right) \\ &= \prod_{i=1}^{n} \left[h\left(\underline{x}_i\right)\right]^{y_i} \times \left[1-h\left(\underline{x}_i\right)\right]^{1-y_i}\end{aligned}$$

The goal is to maximize the log likelihood

$$\log \mathcal{L}(\underline{\theta}) = \sum_{i=1}^{n} \left(y_i \log h\left(\underline{x}_i\right) + (1-y_i) \log\left(1-h(\underline{x}_i)\right)\right), \tag{3}$$

or, alternatively, to minimize the cost function $\mathcal{J}(\underline{\theta})$ that is defined as

$$\mathcal{J}(\underline{\theta}) = -\frac{1}{n} \log \mathcal{L}(\underline{\theta}) \tag{4}$$

How do we minimize the cost function? Similar to our derivation in the case of linear regression, we can use Gradient Descent.

The gradient of the cost function $\mathcal{J}(\underline{\theta})$ is given by

$$\frac{\partial \mathcal{J}(\underline{\theta})}{\partial \theta_j} = \frac{1}{n} \sum_{i=1}^{n} \left[h\left(\underline{x}_i\right) - y_i\right] x_{ij}, \quad j = 1, 2, \ldots, m. \tag{5}$$

and the corresponding matrix form

$$\frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} = \frac{1}{n} \mathbf{X}^T [h(\mathbf{X}) - \underline{y}] \tag{6}$$

then, the gradient descent algorithm updates the parameters using

$$\underline{\theta} = \underline{\theta} - \frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} \tag{7}$$

1.2 Python Exercise

In this part of the exercise, you will implement the logistic regression to predict if a person has cancer or not. We employ the existing dataset from `sklearn` named *breast cancer wisconsin*[1]. This dataset contains in total 569 examples, among them 212 examples are labelled as malignant (M or 0) and 357 examples are marked as benign (B or 1). Features are computed from a digitalized image of a fine needle aspirate (FNA) of a breast mass. The dimension of the feature vector is 30.

In this exercise, we re-use part of the code for the dataset loading and pre-processing from the previous exercises. Thus, we have already completed this part for you.

Step 1: Implement the sigmoid, cost and gradient functions. Similar to the previous exercises, you have to implement the functions `compute_cost` and `compute_gradient`. Also, you need to write code to calculate the output of our hypothesis, namely implement the `sigmoid` function (logistic function).

Step 2: Update the model's parameters using mini-batch gradient descent. In this step, we re-use the implementation of the mini-batch gradient descent algorithm from the previous exercise. You need to write your code to update the model's parameters using the function `compute_gradient` that you have implemented. Again, you have to compute the cost across the training process and visualize it.

Step 3: Predict the probabilities of having cancer and drawing the confusion matrix. In this step, you use the trained model to predict the probabilities of having cancer using the measurements from the test set. You will need to call function `sigmoid` you have implemented before. Moreover, you will evaluate the performance of your model using the accuracy evaluation metric, which is the percentage of correctly classified examples over the total number of examples in the test set. Then, you draw a confusion matrix illustrating your classification result. The accuracy of your model should be greater than 90%

[1] https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

2 Artificial Neural Network

2.1 Theory

In machine learning, artificial neural network is a supervised learning method providing a robust approach to approximating real-valued, discrete-valued and vector-valued target functions. It is one of the most effective methods for certain types of problems such as learning to interpret complex real-world sensor data. Inspired by the biological neural system in the human brain, basically a neural network contains a densely interconnected set of simple units, where each unit takes a number of real-valued inputs and produces a single real-valued output (which may become the input to many other units).

To train a neural network with an optimization method, e.g, gradient descent, we need to know the partial derivatives of our cost function with regard to each network's parameters. For a network with no hidden layer (similar to logistic regression models), this is quite trivial. However, using the same logic with multi-layer networks will only give us the gradient of the last layer. Luckily, backpropagation provides a way to calculate gradient of all the layers using chain rule. To understand more about the intuition of backprogragation, we recommend this tutorial.

2.2 Python Exercise

2.2.1 Exercise 1

In this exercise, you will have to build your neural network, and perform the forward and the backward pass. Then, you will update the network's parameters using gradient descent. In this exercise, we use the digits dataset from sklearn, and some samples are illustrated in Fig. 1. You have to edit the *neural_network.ipynb* file. The architecture of our network is: input $\rightarrow$ fully connected $\rightarrow$ sigmoid $\rightarrow$ fully connected $\rightarrow$ sigmoid $\rightarrow$ output

Step 1: Load, split data and convert labels to one-hot vectors. Like in exercise 1, you will load the dataset and split the data into training and test sets. You also need to add the bias term to the input features, and convert the training output to a one-hot vector.

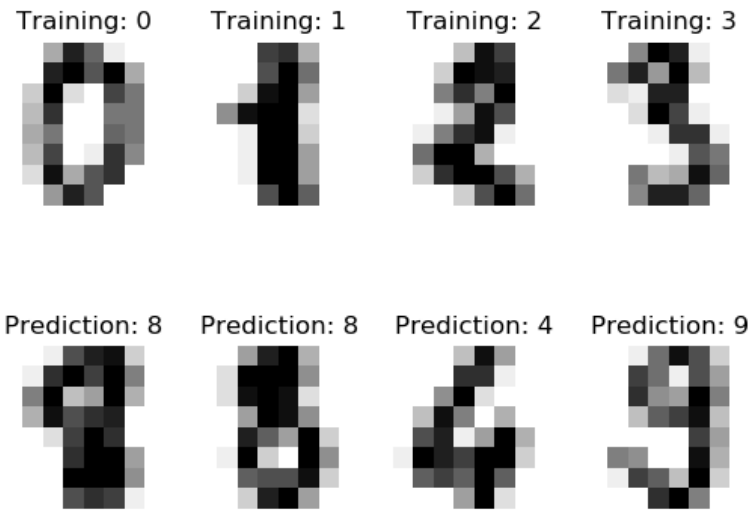

Figure 1: The digit dataset from sklearn.

Step 2: Forward computation. In this step, you will implement several building blocks of a neural network, including the fully-connected layer, the sigmoid activation function and the mean square error cost function. Afterwards, you need to stack each building block sequentially to construct the network architecture described above.

Step 3: Backpropagation. Using backpropagation, you will calculate the gradient for each parameter of the network in this step. First, we need to calculate the gradient for the backward pass for each building block of the network. Afterwards, we traverse the network in reverse order to back-propagate the gradient.

Step 4: Network training. After calculating the forward and backward pass, we will train the network using batch gradient descent. You need to

iterate over your training set several times. For each iteration, we do one forward pass, calculate the loss, and do one backward pass to calculate all the necessary gradients. Then, you update the network's parameters using the gradient descent rule.

Step 5: Evaluation. In this step, you will evaluate the performance of your network on the test set using the accuracy evaluation metric.