1 Logistic Regression

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

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 very poorly. 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, let change the form for our hypotheses $h(\underline{x})$. We will choose

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

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? Following how we saw least squares regression could be derived as the maximum likelihood estimator under a set of assumptions, let endow our classification model with a set of probabilistic assumptions, and then fit the parameters via maximum likelihood. 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 $\mathcal{J}(\underline{\theta})$ 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

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

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. A feature vector has 30 dimensions.

This exercise re-uses the piece of code for dataset loading and pre-processing from the previous exercises. Therefore, this part has been done 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). At the end of this step, you will use the function `approximate_gradient` to verify if your implementation is correct.

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 caner and drawing the confusion matrix. In this step, you use the trained model to predict the probabilities of having caner 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 accuracy measure, 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 k-Nearest Neighbor Classification

2.1 Theory

k-Nearest-Neighbor (kNN) classification is one of the most fundamental and simple classification methods and should be one of the first choices for a classification study when there is little or no prior knowledge about the distribution of the data. kNN classification was developed from the need to perform discriminant analysis when reliable parametric estimates of probability densities are unknown or difficult to determine.

The kNN classifiers are memory-based, and require no model to be fit. Given a query point x_0, we find the k training points x_i, $i = 1 \dots k$ closest in distance to x_0, and then classify using majority vote among the k neighbors. For a deeper theoretical understanding of the kNN algorithm see:

- Section 13.3, Ch. 13 from the book "The Elements of Statistical Learning" of Hastie, Tibshirani and Friedman.

2.2 Python Exercise

In this exercise, we will work with a sample dataset from *sklearn* containing images of digits, from 0 to 9. We will implement a kNN classifier to classify each sample into one of the 10 classes.

Step 1: Load the dataset. Use the function *load_digits* from *sklearn* to load the dataset.

Step 2: Split data. Split the original dataset into training and testing sets, so that the training set accounts for 80% of the samples. Don't forget to split both the features, i.e. X and the targets, i.e. y.

Step 3: Euclidean distance function. Write a Python function to calculate the Euclidean distance between each pair of rows of two input matrices. Given the input matrices as $A \in \mathbb{R}^{n\times d}$ and $B \in \mathbb{R}^{m\times d}$, your function needs to return the distances in form of a matrix of shape $n \times m$.

Note:

- Start with explicit implementation in this step (using for loops to process all possible pairs). We will write a vectorized implementation in Step 7

- You need to implement this function using basic Numpy operators, such as *sum, square, sqrt*.

- Do not call built-in function from Numpy for this task.

Step 4: Write evaluation function. Write a Python function to compare your predictions to the target labels and calculate the average accuracy. This function takes as input two Numpy arrays and return a single floating-point value.

Step 5: kNN classification. Implement a simple kNN classifier, calling your Euclidean distance function to calculate distances from each test samples to all the training samples. Based on the calculated distances, select for each test sample k nearest neighbors from the training set and find the prediction for it by doing majority voting.

Try different values of k to see how the accuracy changes.

Step 6: Vectorized Euclidean distance function. In Numpy, writing vectorized functions instead of explicit ones (with lots of for loops) can significantly improve the code efficiency and speed. This applies to other libraries, such as Matlab as well.

In this step, write another Python function to calculate the Euclidean distance between each pair of rows of two input matrices. The signature of this function is the same as the one in Step 3.

Note:

- You need to implement this function using basic Numpy operators, such as *sum, square, sqrt*.

- Do not call built-in function from Numpy for this task.

- Correct implementation does not have any for loop.

Step 7: Vectorized kNN classification. Re-implement the kNN classifier in Step 5 in vectorized form. You should call your vectorized Euclidean distance function, and write as few for loop as possible. Evaluate your accuracy in this step to see if it's the same as in Step 5.