# 4 Vectorized implementation in Numpy

### 4.0.1 Short Theory

In the previous chapters, you have been extensively using the Numpy library, which allows efficient matrix and algebraic operations in Python. In this chapter, we will look at an essential feature in order to exploit Numpy's full power, namely: *vectorized implementation*. This will probably sound familiar for those who have already programmed in Matlab as the same principles hold. Both Numpy and Matlab have been optimized to be extremely efficient in handling matrix operations. As for-loops take much more time in comparison to those matrix operations, we can describe the golden rule as "*writing as few for-loops as possible*"

As a demonstration of the vectorized implementation, we will consider the problem of pairwise Euclidean distance calculation between two matrices. Suppose we have two matrices, $A \in \mathbb{R}^{n\times d}$ and $B \in \mathbb{R}^{m\times d}$. We can think of $A$ and $B$ as containing respectively $n$ and $m$ samples of $d$ dimensions. The task is to compute a distance matrix $D \in \mathbb{R}^{n\times m}$, where each element $d_{ij}$ for $i \in \{1,\dots,n\}$ and $j \in \{1,\dots,m\}$, is the Euclidean distance between $\underline{a}_i \in \mathbb{R}^{dx1}$ ($i-th$ row of $A$) and $\underline{b}_j \in \mathbb{R}^{dx1}$ ($j-th$ row of $B$). This distance calculation task is the center of many machine learning algorithm, such as *K-means clustering*, *K-nearest neighbor classification* and *'ranking and retrieval'*.

Recall that the Euclidean distance between vector $\underline{a}_i$ and $\underline{b}_j$ is calculated as follows:

$$\begin{aligned} d_{ij} &= \sqrt{(a_{i,1}-b_{j,1})^2+(a_{i,2}-b_{j,2})^2+\cdots+(a_{i,d}-b_{j,d})^2} \\ &= \sqrt{\sum_{k=1}^{d}(a_{i,k}-b_{j,k})^2} \end{aligned} \tag{19}$$

**Straightforward implementation with for loops**

A straightforward way to implement a function performing this task is to loop over all pairs of row vectors between the two matrices $A, B$ and calculate their distances using eq.(19). This approach is simple, but it is not efficient (in Numpy) as it does not utilize the highly optimized vector/matrix operations in Numpy.

**Vectorized implementation**

Expand eq. (19) further, we have:

$$\begin{aligned} d_{ij} &= \sqrt{\sum_{k=1}^{d} (a_{i,k})^2 - 2a_{i,k}b_{j,k} + (b_{j,k})^2} \\ &= \sqrt{\left(\sum_{k=1}^{d} (a_{i,k})^2\right) + \left(\sum_{k=1}^{d} (b_{j,k})^2\right) - 2\left(\sum_{k=1}^{d} a_{i,k}b_{j,k}\right)} \\ &= \sqrt{\|\underline{a}_i\|_2^2 + \|\underline{b}_j\|_2^2 - 2\underline{a}_i^T\underline{b}_j} \end{aligned} \tag{20}$$

As eq. (20) suggests, in order to calculate the distance between the row `i` in A to the row `j` in B, we need to calculate the length of $A_i$, the length of $B_j$ and their inner product. A simple way to do it over all pairs of rows in A and B is, again, performing `for` loop over all `i` and `j`. However, a better way is to (i) first calculate the lengths of all rows in A, the length of all rows in B and the inner product between all pairs of rows in A and B; and (ii) sum them over and take the element-wise square root to get the distance matrix D.

### 4.0.2 Python Exercise

In this exercise, you will implement and experiment with the two approaches to compute the distance matrix D as presented above.

**Step 1: Euclidean distance calculation with `for` loop.** Employ the straightforward approach, using for-loops, to calculate the distance matrix D.

**Step 2: Vectorized implementation of Euclidean distance calculation.** Employ the vectorized approach to calculate the distance matrix D. Note that a correct implementation should not contain any `for` loop.

**Step 3: Checking the correctness.** Either approach you follow, you should get the same results for the distance matrix D (though this does not necessarily means you have correct results). Run the code cell to check this.

**Step 4: Comparing the running time.** Run the code cell and compare the difference in running time between the two implementations. The vectorized implementation should be significantly faster than the straightforward implementation. The speed-up

factor depends on the size of the matrices. When working with big matrices, you can have a very large speed gain using the vectorized approach.

# 5 Regression 2.0

## 5.1 LASSO Regression

### 5.1.1 Theory

In this exercise, we will work with the LASSO regression model, which is a linear model equipped with a L1-regularization term. Recall that we have worked with a regularized linear regression model in both chapters 2 and 3. The employed model was a (slight variation of) Ridge models, as they were equipped with an l2-regularization term. The optimization objective for LASSO regression is:

$$L_{LASSO}(X;\underline{\theta}) = \frac{1}{2 \times N_{samples}}||\underline{y} - X\underline{\theta}||_2^2 + \lambda||\underline{\theta}||_1. \tag{21}$$

$\lambda$ is often referred to as the *regularization parameter*.

### 5.1.2 Python Exercise

In this exercise, you will, again, work with the Boston house price dataset from the sklearn library. The goal is to implement and make experiments with the LASSO model to solve the problem of predicting house prices given their descriptive information (or features). At the end of the exercise, you will compare the performance of the LASSO model with that of a non-regularized linear model and see the effect of the L1-regularization.

For the implementation, you can make use of the APIs provided by the sklearn library for the LASSO regression model[7]. The steps that you need to follows are listed below:

**Step 1: Preprocess the data and add the intercept term.** Load the Boston dataset from sklearn and split it to an 80%-20% training-testing ratio. Then, scale all the features to a similar value range, and add the intercept term to the train and the test data. The step is the same as in other exercises in 3, so you can reuse your code.

**Step 2: Train the LASSO regression model.** Instantiate a LASSO object (using the sklearn library), and train the model using the prepared training set. As a starting point, you can use $\lambda = 0.1$.

**Step 3: Predict the price with the LASSO model.** Use the trained model to make predictions for samples on the test set, then calculate the prediction errors in terms of the Mean Absolute Error (MAE) and Mean Squared Error (MSE). The functions in

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

sklearn.metrics can be used to calculate these metrics.

**Step 3: Compare the performance with the linear regression model.** Train a linear regression model to use as a reference for performance comparison. Make predictions and estimate the MAE and MSE. Compare the results with the errors given by the LASSO model.

**Step 4: Select the best lambda.** In this step, you need to use different $\lambda$-values to re-train the model. By observing the MAE and MSE prediction error, you can select the most appropriate $\lambda$-value, leading to the best results on the test set. A possible test range for $\lambda$ is $(0, 0.5]$.

## 5.2 Logistic Regression

While having focused on regression problems in the previous chapters, we will now examine a specific classification problem. One could approach a classification problem as a regression problem with the extra constraint that the values $y$ should be part of a small number of discrete values. For now, we will focus on the **binary** classification problem in which $y$ can only be two values, 0 and 1. For instance, if we are trying to build an e-mail spam classifier, we could define $\underline{x}_i$ as a set of features of an e-mail, while $y$ classifies it in spam ($y = 1$) and ham[8] ($y = 0$). 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 sample $\underline{x}_i$, the corresponding $y_i$ is also called its **label**.

### 5.2.1 Theory

We could approach the classification problem ignoring the fact that $y$ is discrete-valued, and use our linear regression algorithms,

$$y_i = h_{\text{lin\_regression}}(\underline{x}_i) = \underline{x}_i\underline{\theta}, \tag{22}$$

to try and predict $y_i$ given $\underline{x}_i$. However, this will often perform very poorly. Intuitively, it does not make sense that our hypothesis $h(\underline{x}_i)$ allows an output larger than 1 or smaller than 0 as we know that $y_i \in [0, 1]$. To resolve this issue, we can introduce the following non-linear function,

$$g(z) = \frac{1}{1 + e^{-z}}, \tag{23}$$

[8]E-mails which are not spam, are called 'ham', as introduced in a Monty Python sketch

which is called the **logistic** (or the **sigmoid**) function. Notice that $g(z)$ tends towards 0 as $z \to -\infty$, and $g(z)$ tends towards 1 as $z \to \infty$. When applying it on our previous hypothesis, we obtain

$$h(\underline{x}_i) = g\left(\underline{x}_i \underline{\theta}\right) = \frac{1}{1 + e^{-\underline{x}_i \underline{\theta}}}. \tag{24}$$

Since $g(z)$ is bounded between 0 and 1, $h(\underline{x}_i)$ is too.

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 $P(y_i = 1|\underline{x}_i; \underline{\theta}) = h(\underline{x}_i)$ and $P(y_i = 0|\underline{x}_i; \underline{\theta}) = 1 - h(\underline{x}_i)$. Note that this can be written more compactly as

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

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

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

The goal is to maximize the log likelihood

$$\log \mathcal{L}(\underline{\theta}) = \sum_{i=1}^{n} \left(y_i \log h\left(\underline{x}_i\right) + (1 - y_i) \log\left(1 - h(\underline{x}_i)\right)\right), \tag{25}$$

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

$$\mathcal{J}(\underline{\theta}) = -\frac{1}{n} \log \mathcal{L}(\underline{\theta}) \tag{26}$$

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

$$\frac{\partial \mathcal{J}(\underline{\theta})}{\partial \theta_j} = \frac{1}{n} \sum_{i=1}^{n} \left[h\left(\underline{x}_i\right) - y_i\right] x_{ij} \quad \text{for } j = 1, 2, \ldots, m. \tag{27}$$

and the corresponding matrix form

$$\frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} = \frac{1}{n} X^T [h(X) - \underline{y}] \tag{28}$$

then, the gradient descent algorithm updates the parameters using

$$\underline{\theta} = \underline{\theta} - \frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} \tag{29}$$

### 5.2.2 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*[9]. 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 and drawing the confusion matrix.** In this step, you use the trained model to predict the probabilities 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%

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