Chapter 1

Training Simple Machine Learning Algorithms for Classification

So, we will see basic algorithms for classification and some other useful things.

1.1 Introduction

In this section we will make use of the first two algorithmically described machine learning algorithms for classification: the **perceptron** and **adaptive linear neurons**.

Frist, we'll start by implementing a perceptron step by step in Python and training it to classify different flower species in the iris dataset.

Observation 1.1.1
Using this algorithm and discussing the basics of optimization using adaptive lienar neurons layse the groundwork for using more sophisticated classifiers.

The goals of this chapter are the following:

- Building an understanding of machine learning algorithms
- Using pandas, NumPy, and Matplotlib to read in, process, and visualize data
- Implementing linear classifiers for 2-class problems in Python

1.2 Artificial Neurons: Early Stages of Machine Learning

Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell, the so-called McCulloch-Pitts (MCP) neuron, in 1943 (A Logical Calculus of the Ideas Immanent in Nervous Activity by W. S. McCulloch and W. Pitts, Bulletin of Mathematical Biophysics, 5(4): 115-133, 1943).

Observation 1.2.1 (Biological Neurons)
Biological Neurons are interconnected nerve cells in the brain that are involved in the processing and transmiting of chemical and electrical signals.

Idea 1.2.1
They described a nerve cell as a **simple logic gate with binary outputs**.

Multiple signals arrive at the dendrites, they are then integrated into the cell body and, if the accumulated signals exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

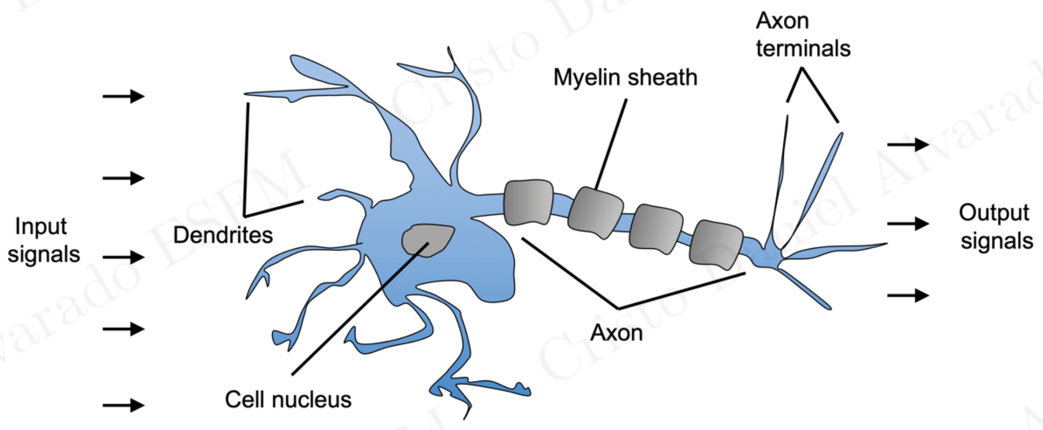

Figure 1.1: Biological Neuron.

Observation 1.2.2 (**Perceptron Learning Rule**)
Only a few years later, Frank Rosenblatt published the first concept of the **perceptron learning rule based** on the MCP neuron model (*The Perceptron: A Perceiving and Recognizing Automaton* by F. Rosenblatt, Cornell Aeronautical Laboratory, 1957). With his perceptron rule, Rosenblatt *proposed an algorithm that would automatically learn the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not.*

In the context of supervised learning and classification, *such an algorithm could then be used to predict whether a new data point belongs to one class or the other.*

1.3 Formal Definition of an Artificial Neuron

We can put the idea of **articial neurons** into the context of a binary classification task with two classes: 0 and 1.

Definition 1.3.1 (**Decision Function**)
A **decision function** is a function $\sigma : X \to \{0,1\}$ that takes a linear combination of a certain input values (called x) and a corresponding weight vector (called w) , where z is the net input:

$$z = w_1x_1 + w_2x_2 + \cdots + w_mx_m = \sum_{i=1}^{m} w_ix_i$$

so, $\sigma(z) \in \{0,1\}$.

If the net input of a particular example is greater than a defined threshold (lets say $\vartheta \in \mathbb{R}$, so $z \geq \vartheta$), we predict class 1, and class 0 otherwise. In the perceptron algorithm, the decision function, $\sigma(z)$ is a variant of a **unit step function**:

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq \vartheta \\ 0 & \text{otherwise} \end{cases}$$

Observation 1.3.1
We can modify the setup with a couple of steps. So, we make the **bias unit** $b = -\vartheta$, and we make:

$$z' = w_1 x_1 + w_2 x_2 + \cdots + w_m x_m + b = \vec{w}^T \vec{x} + b$$

And now replacing every input z by z' we obtain that:

$$\sigma(z') = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Observation 1.3.2
The transpose A^T of matrix $A \in \mathcal{M}_{m \times n}$ is defined as:

$$A^T = \begin{bmatrix} a_1^{(1)} & a_2^{(1)} & \cdots & a_n^{(1)} \\ a_1^{(2)} & a_2^{(2)} & \cdots & a_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ a_1^{(m)} & a_2^{(m)} & \cdots & a_n^{(m)} \end{bmatrix}$$

where:

$$A = \begin{bmatrix} a_1^{(1)} & a_2^{(1)} & \cdots & a_m^{(1)} \\ a_1^{(2)} & a_2^{(2)} & \cdots & a_m^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ a_1^{(n)} & a_2^{(n)} & \cdots & a_m^{(n)} \end{bmatrix}$$

Observation 1.3.3 (**Notation Problem**)
This notation is horrible and differs from the one Mathmaticians use in all his textbooks, please, don't use this notation.

1.4 Perceptron Learning Rule

The whole idea behind the MCP neuron and Rosenblatt's thresholded perceptron model is to *use a reductionist approach to mimic how a single neuron in the brain works*: **it either fires or it doesn't**. Thus, Rosenblatt's classic perceptron rule is fairly simple, and the perceptron algorithm can be summarized by the following steps:

1. Initialize the weights and bias unit to 0 or small random numbers
2. For each training example, $x^{(i)}$.
3. Compute the output value, $\hat{y}^{(i)}$.
4. Update the weight and bias unit.

Observation 1.4.1 (**Output Value**)
Here, the **output value** is the **class label** *predicted by the unit step function defined earlier.*

The simultaneous update of each weight w_j in the weight vector $\vec{w}$ can be formally written as:

$$w_j \equiv w_j + \Delta w_j$$

and,

$$b \equiv b + \Delta b$$

Observation 1.4.2 (**Use of the Symbol** $\equiv$)
Here, the symbol $\equiv$ states that the vlaue of the variable to the left is updated to that of the variable on the right.

Where, the updated values (or **deltas**) are computed as follows:

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

and,

$$\Delta b = \eta(y^{(i)} - \hat{y}^{(i)})$$

Observation 1.4.3 (**Notation Abuse**)
This thing is absoultely ridicolus in terms of notation, the correct notation should be something like $\Delta w_j^{(i)}$, but since this variable is not going to be used anymore, we simply forgot about it, this is because in the process this variable is going to be updated every single time and we dont care about the different values depending on the i.

Here, η is a learning rate (usually between 0 and 1). $y^{(i)}$ is the **true class label** and $\hat{y}^{(i)}$ is the **predicted class label**.

Observation 1.4.4
The bias unit and all weights in the weight vector are updated simultaneously, which means that the predicted label $\hat{y}^{(i)}$ is not recomputed before the bias unit and all the wrights are updated via the respective update values, ,Δw_j and Δb.

Example 1.4.1
Concretely, for a two-dimensional dataset, we would write the update as:

$$\Delta w_1 = \eta(y^{(i)} - \text{output}^{(i)})x_1^{(i)}$$
$$\Delta w_2 = \eta(y^{(i)} - \text{output}^{(i)})x_2^{(i)}$$
$$\Delta b = \eta(y^{(i)} - \text{output}^{(i)})$$

1. Before we implement the perceptron rule in Python, let's go through a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the bias unit and weights remain unchanged, since the update values are 0:

2. **When prediction is correct:**

   - If $y^{(i)} = 0$ and $\hat{y}^{(i)} = 0$:

$$\Delta w_j = \eta(0 - 0)x_j^{(i)} = 0$$
$$\Delta b = \eta(0 - 0) = 0$$

   - If $y^{(i)} = 1$ and $\hat{y}^{(i)} = 1$:

$$\Delta w_j = \eta(1 - 1)x_j^{(i)} = 0$$
$$\Delta b = \eta(1 - 1) = 0$$

3. **When prediction is wrong:**
   - If $y^{(i)} = 0$ and $\hat{y}^{(i)} = 1$:

$$\Delta w_j = \eta(0-1)x_j^{(i)} = -\eta x_j^{(i)}$$
$$\Delta b = \eta(0-1) = -\eta$$

   - If $y^{(i)} = 1$ and $\hat{y}^{(i)} = 0$:

$$\Delta w_j = \eta(1-0)x_j^{(i)} = \eta x_j^{(i)}$$
$$\Delta b = \eta(1-0) = \eta$$

4. To get a better understanding of the feature value as a multiplicative factor $x_j^{(i)}$, consider another example where $y^{(i)} = 1$, $\hat{y}^{(i)} = 0$, $\eta = 1$. Assume that $x_j^{(i)} = 1.5$ and this example is misclassified as class 0. In this case, we would increase the corresponding weight so that the net input $z = x_j^{(i)} w_j + b$ would be more positive the next time we encounter this example and thus more likely to be above the threshold of the unit step function to classify the example as class 1:

$$\Delta w_j = (1-0) \times 1.5 = 1.5$$
$$\Delta b = (1-0) = 1$$

5. The weight update Δw_j is proportional to the value of $x_j^{(i)}$. For instance, if we have another example $x_j^{(i)} = 2$ that is incorrectly classified as class 0, we will push the decision boundary by an even larger extent to classify this example correctly the next time:

$$\Delta w_j = (1-0) \times 2 = 2$$
$$\Delta b = (1-0) = 1$$

6. The convergence of the perceptron is only guaranteed if the two classes are linearly separable, which means that the two classes can be perfectly separated by a linear decision boundary. If the two classes cannot be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (epochs) and/or a threshold for the number of tolerated misclassifications—the perceptron would never stop updating the weights otherwise.

7. The perceptron receives the inputs of an example (x) and combines them with the bias unit (b) and weights (w) to compute the net input. The net input is then passed on to the threshold function, which generates a binary output of 0 or 1—the predicted class label of the example. During the learning phase, this output is used to calculate the error of the prediction and update the weights and bias unit.

1.5 Implementing a Perceptron Learning Algorithm in Python