Machine Learning with PyTorch and Scikit-Learn From Coursera

Cristo Daniel Alvarado

October 27, 2025

Contents

Code List

CHAPTER 1

MODULE 1

The goal of this chapter is to explore the foundational concepts of machine learning, focusing on how algorithms can transform data into knowledge. We delve into the practical applications of supervised and unsupervised learning, equipping you with the skills to implement these techniques using Python tools for effective data analysis and prediction.

> **Observation 1.0.1 (Learning Objectives)**
> **Learning Objectives**:
>
> - Analyze data patterns to make future predictions.
>
> - Design systems for supervised and unsupervised learning.
>
> - Implement machine learning algorithms using Python tools.

1.1 OVERVIEW

The goal basically is to learn machine learning using Python libraries such as Scikit-Learn and PyTorch.

1.2 GIVING COMPUTERS THE ABILITY TO LEARN FROM DATA

So, basically in this scenario we will do the following:

- Implement machine learning algorithms using Python tools.

- Desing systems for supervised and unsupervised learning.

- Analyze data patterns to make future predictions.

1.3 INTRODUCTION

> **Definition 1.3.1 (Machine Learning)**
> **Machine learning** is the *application and science of algorithms that make sense of data.*

The goal of this section is to introduce some of the main concepts and different types of machine learning, together with a basic introduction to the relevant terminology. The goal is to stablish the groundwork for successfully using machine learning techniques for practical prolem solving.

1.3.1 The Three Different Types of Machine Learning

In this age, we have a large amount of structured and non-structured data.

> **Definition 1.3.2 (Structured and Non-structured Data)**
> **Structured data** refers *to information that is organized in a predefined format*, typically within a fixed schema, such as rows and columns in a database or spreadsheet
>
> In contrast, **non-structured data** *lacks a predefined format or structure and exists in its native, raw state.* It is typically qualitative and encompasses a wide variety of formats such as text documents, emails, social media posts, images, videos, and audio files.

One of the main goals of machine learning is to extract knowledge from data in order to make predictions.

> **Observation 1.3.1 (Use of Machine Learning)**
> **Machine Learning** *offers a more efficient alternative for capturing the knowledge in data to improve the performance of predictive models and make data-driven decisions.*

> **Idea 1.3.1 (Note on the use of Machine Learning and some of its Applications)**
> Also, notable progress has been made in medical applications; for example, researchers demonstrated that deep learning models can detect skin cancer with near-human accuracy link to article.
>
> Another milestone was recently achieved by researchers at DeepMind, who used deep learning to predict 3D protein structures, outperforming physics-based approaches by a substantial margin link to article.

There are three types of machine learning: **supervised learning**, **unsupervised learning**, and **reinforcement learning**. There are fundamental differences between these three types of machine learning and we will look at each of them in detail with some notes on its possible applications.

Three Types of Machine Learning	
Supervised learning	• Labeled data • Direct feedback • Predict outcome/future
Unsupervised learning	• No labels/targets • No feedback • Find hidden structure in data
Reinforcement learning	• Decision process • Reward system • Learn series of actions

Table 1.1: Three Types of Machine Learning.

1.3.2 Supervised Learning

The main goal of **supervised learning** is to *learn a model from labeled training data that allows us to make predicitons about unseen or future data.*

The term *supervised* refers to a set of training examples (data inputs), where the desired output signals (labels) are already known.

Definition 1.3.3 (Supervised Learning)
Supervised learning is the *process of modeling the relationship between data inputs and the labels.*

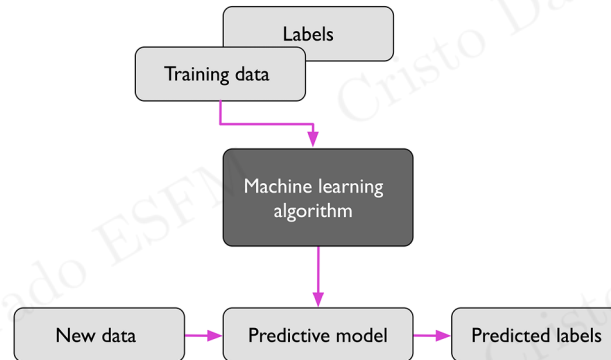We can think of supervised learning as *label learning*.

Figure 1.1: General Process of Supervised Learning

Example 1.3.1 (Example of Supervised Learning)
Lets consider the example of email spam filtering. We can train a model using a supervised machine learning algorithm on a corpus of labeled emails, which are correctly marked as spam or non-spam.

To predict whether a new email belongs to either of the two categories, a supervised learning task with discrete class labels, such as in the previous email spam filtering example.

This is called a **classification task**. Another subcategory of supervised learning is **regression**, where the outcome signal is a continous value.

1.3.3 CLASSIFICATION FOR PREDICTING CLASS LABELS

Definition 1.3.4 (Classification)
Classification is *a subcategory of supervised learning, where the goal is to predict the categorical class labels of new instances or data points based on past observations.*

Thos class labels are discrete, unordered values that can be understood as the group memberships of the data points. The previously mentioned example of email spam detection represents a typical example of a binary classification task, where the machine learning algorithm learns a set of rules to distinguish between two possible classes: spam and no-spam emails.

The next ilustration shows the concept of binary classification task given 30 training examples; 15 training are labeled as **class A** and the other 15 labeled as **class B**.

Our dataset is two-dimensional, meaning that each example has two values associated with it: x_1 and x_2. We can use a supervised machine learning algorithm to to learn a rule (decision boundary represented as the dashed line) that can separate two classes and classify new data into each of those two categories given its x_1 and x_2 values.
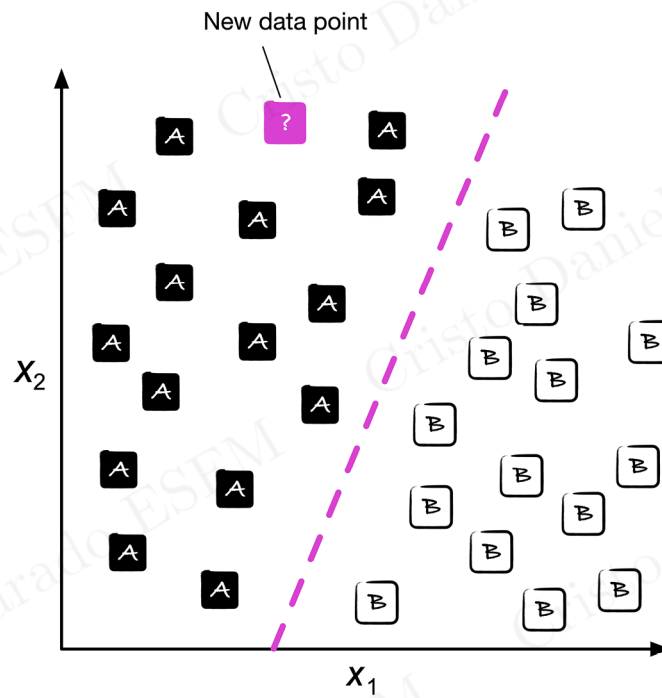
Figure 1.2: Classification in Machine Learning

Observation 1.3.2 (Nature of classification of class labels)
When a set of class labels does not have to be of a binar ynature. The predictive model learned by a supervised learning algorithm can assign any class label that was presented in the training dataset to a new, unlabeled data point or instance.

Example 1.3.2
An example of a multiclass classification task is handwritten character recognition.

1.3.4 Regression for Predicting Continous Outcomes

Definition 1.3.5 (Regression Analysis)
In a **regression analysis**, *we are given a number of predictor (**explanatory**) variables and a continus response variable (**outcome**) and we try to finde a relationship between those variables that allows us to predict an outcome.*

Observation 1.3.3 (Feature and Target Variables)
In the field of machine learning, the predictor variables are commonly called *features*, and the response variables are referred to as *target variables*.

Example 1.3.3
Let's asume we want to predict the mat SAT scores of students. If there is a relationship between time spent studying for the test and the final scores, we could use it as training data to learn a model that uses the study time to predict the test scores of future students.

7

Given a feature variable x and a target variable y, we fit a straight line to the data points that minimizes the error in predicting y from x.

We can now use the intercept and slope learned from this data to predict the target variable of new data.
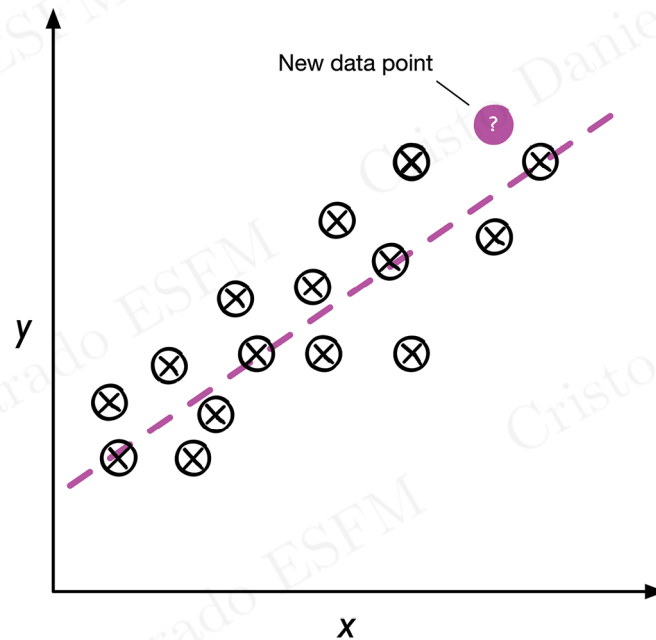
Figure 1.3: Linear Aproximation

1.4 Solving Problems with Reinforcement Learning

Definition 1.4.1 (Reinforcement Learning)
Reinforcement Learning has the *goal to develop a system* (called **agent**) *that improves its performance through interactions with the enviroment.*

Observation 1.4.1
Typically, the information coming from a current state of the enviroment also includes a **reward signal**. With this in mind, reinforcement learning can be vewed as being realted to supervised learning.

This feedback is not the correct ground-truth label or value, *but a measure of how well the action is evaluated by a reward function.*

Using a reward signal, a system can measure this reward via an exploratory trial-and-error approach or deliberative planning.

Example 1.4.1
One of the most popular examples of reinforcement learning is **chess program**. The agent decides upon a series of moves and the reward can be defined as win or lose at the end of the game.

In synthesis, the reinforcement learning can be described with the following diagram.

Figure 1.4: Reinforcement Learning Process

There are many different subtypes of reinforcement learning, but in a general scheme the agent tries to maximize the reward through a series of interactions with the enviroment.

> **Idea 1.4.1**
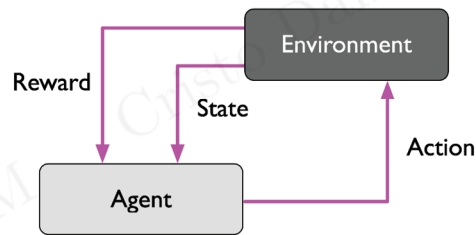> In summary, reinforcement learning is concerned with learning to choose a series of actions that maximizes the total reward which could be earned immediately after taking an action or vía *delayed* feedback.

1.5 UNSUPERVISED LEARNING

Unsupervised learning tries to find the hidden structures of our data. Using supervised learning or reinforcement learning, we have the right answer or a measure of a reward for particular actions, respectively. Unsupervised learning explores the structure of our data to extract meaningful information without the guidance of a known outcome variable.

1.5.1 FINDING SUBGROUPS WITH CLUSTERING

> **Definition 1.5.1 (Clustering)**
> **Clustering** *is an exploratory data analysis or pattern discovery technique that organizes information into meaningful subgroups* (called **clusters**) *without any prior knowledge of group memberships.*

Each cluster defines a group of objects that share a certain degree of similarity but are more dissimilar to objects in other clusters which is why clustering is called (sometimes) **unsupervised classification**.

The following scatter plot shows how clustering can organizse unlabeled data into three disctinct groups or clusters.

1.5.2 DIMENSIONALITY REDUCTION FOR DATA COMPRESION

Another subfield of unsupervised learning is dimensionality reduction.

> **Definition 1.5.2 (Dimensionality Reduction)**
> **Dimensionality reduction** is the process to reduce the dimension of the data we are working it, preserving the original and meaningful data needed to interpret it.

Often, we work with data of high dimensionality that can present challenges for limited storage space and the computational performance of machine learning algorithms. The unsupervised deimensionality reduction is commonly used in feature preprocessing to remove noise from data, which can degrade the predictive performance of certain algorithms.
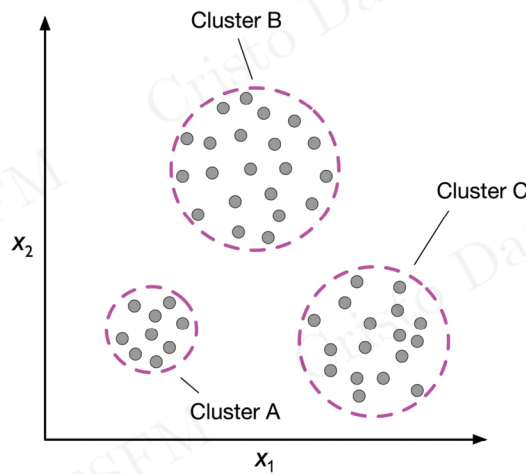
Figure 1.5: Caption

Dimensionality reduction compresses the data into a smaller dinensional subspace while retaining most of the relevant information.

> **Observation 1.5.1**
> Sometimes, dimensionality reduction is useful for visualizing data. The following figure is a good and useful example of it.
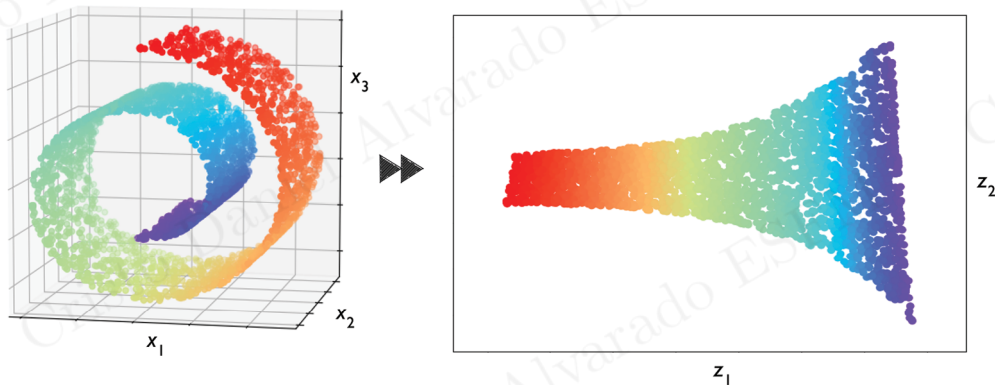
Figure 1.6: Example of Dimensionality Reduction in a Dataset.

1.6 TERMINOLOGY AND NOTATIONS

> **Definition 1.6.1 (Dataset)**
> A **dataset** is a *collection of related sets of information that is composed of separate elements* but can be manipulated as a unit by a computer.

One image that contains most of the information that we'll be using throught this course is the following:
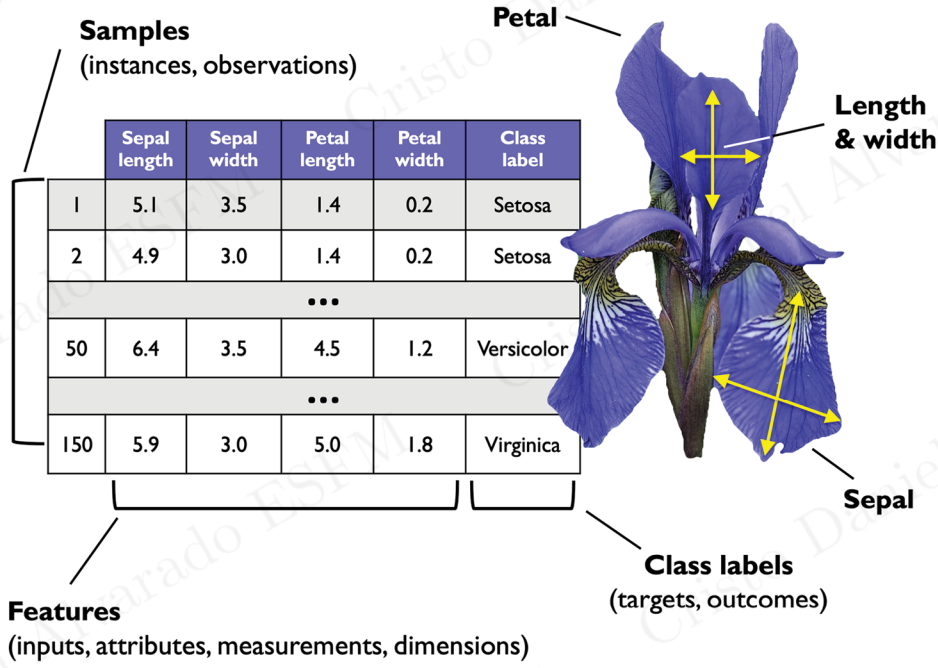
Figure 1.7: Samples, Features, and Class labels.

Example 1.6.1
The Iris dataset contains the measurements of 150 iris flowers from three different species (setosa, versicolor and virginica).

Each flower example represents one row in our dataset and the flower measurements in centimeters are stored as columns, called **features of the dataset**.

Definition 1.6.2 (Feature)
In a dataset, a **feature** *is an input, attribute, measurement or dimensions of something.*

Idea 1.6.1 (Notation)
The iris dataset consists of 150 examples and four features, we will write them as a 150×4 matrix:

$$
X = \begin{bmatrix}
x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\
x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\
\vdots & \vdots & \vdots & \vdots \\
x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)}
\end{bmatrix}
$$

Vectors will be identified as the column matrix $\vec{x} \in \mathbb{R}^{n \times 1}$. To refer to the elements of the matrix we will use the notation $x_i^{(j)}$.

Observation 1.6.1 (Use of Notation)
In the latter idea, $x_1^{(150)}$ refers to the first dimension flower example 150, corresponding to the sepal length.

Each row in the matrix X represents one flower instance and can be written as a four-

dimensional row vector:
$$x^{(i)} = \left[\begin{array}{cccc} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{array} \right]$$

Each feature is a 150-dimensional column vector, denoted by x_j:

$$x_j = \left[\begin{array}{c} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{array} \right]$$

We represent the target variables (here, class labels) as 150-dimensional column vector:

$$y = \left[\begin{array}{c} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(150)} \end{array} \right]$$

where $y^{(i)} \in \{\text{Setosa}, \text{Versicolor}, \text{Virginica}\}$.

1.6.1 Machine Learning Teminology

Definition 1.6.3 (Terminology in Machine Learning)
Some of the following terms are largely used in machine learning:

- **Training example**: A row in a table representing the dataset and synonymous with an observation, record, instance, or sample (in most contexts, sample refers to a collection of training examples).

- **Training**: Model fitting, for parametric models similar to parameter estimation.

- **Feature** (x): A column in a data table or data (design) matrix. Synonymous with predictor, variable, input, attribute, or covariate.

- **Target** (y): Synonymous with outcome, output, response variable, dependent variable, (class) label, and ground truth.

- **Loss function**: Often used synonymously with a cost function. Sometimes the loss function is also called an error function. In some literature, the term 'loss' refers to the loss measured for a single data point, and the cost is a measurement that computes the loss (average or summed) over the entire dataset.

1.7 Machine Learning Systems

In this section we'll dsicuss the other important parts of a machine learning system accompaying the learning algorithm.

1.7.1 Preprocessing: Getting Data into Shape

Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm. Thus, the preprocessing of the data is one of the most crucial steps in any machine learning application.
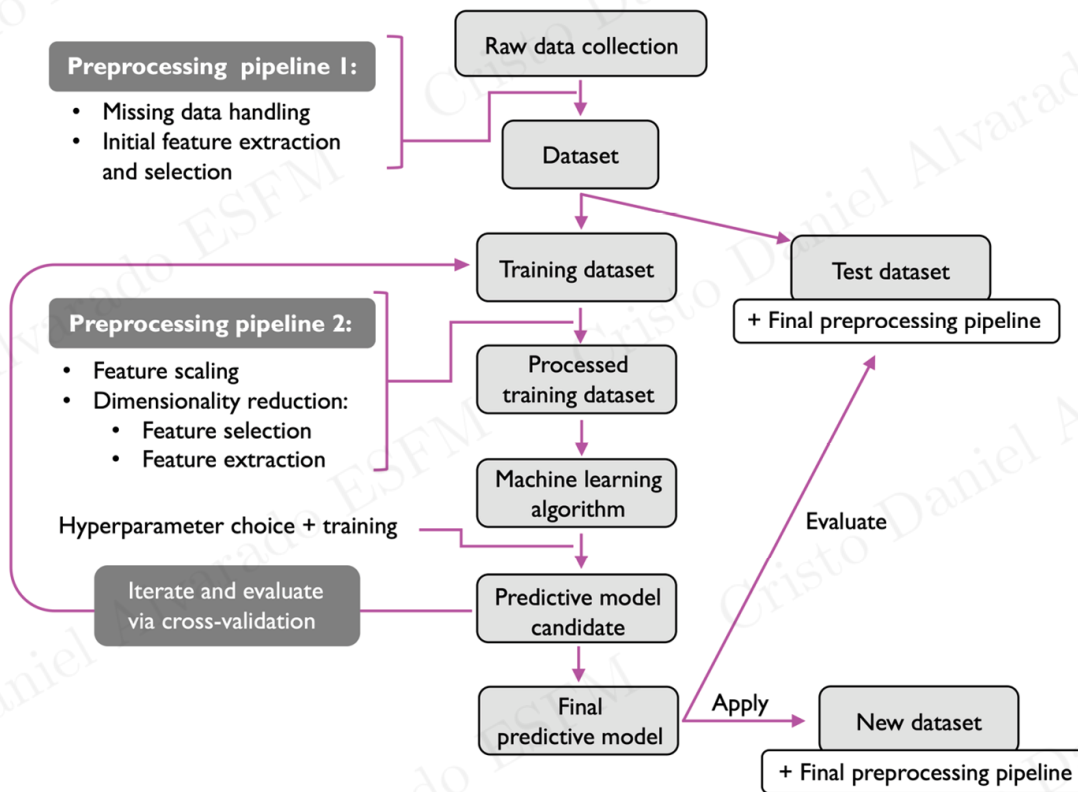
Figure 1.8: Machine Learning System.

Definition 1.7.1 (Preprocessing)
Preprocessing is the *process of converting raw data into dara suitable for a machine learning algorithm.*

Example 1.7.1 (Optimal Features in the Iris Dataset)
Let's consider the iris dataset from the previous example. We can think of raw data as a series of flower images from which we want to extract **meaningful features**. Useful features could be:

- Centered around the color of the flowers.

- Height.

- Length.

- Width.

Observation 1.7.1 (Data Convesion)
Many machine learning models require that the selected features are on the same scale for optimal performance, which is achieved by transforming the features to the range $[0, 1]$.

Other way to achieve this is use standard normal distribution with zero mean and unit variance.

Observation 1.7.2 (Use of Dimensionaly Reduction)
When certain features are highly correlated and therefore, redundant, dimensionaly techniques may be useful for compressing the features onto a lower dimensional subspace.

The main advantage of dimensionality reduction is that less storage space is required, and therefore the learning algorithm can run much faster. This can also improve the performance of a model if the dataset contains a large number of irrelevant features (or noise).

Definition 1.7.2 (Noise)
In data science and statistics, **noise** refers to the *random, uncontrollable, and unexplained variations in the data. It is the part of the data that does not represent the underlying phenomenon you are trying to study or model.*

Observation 1.7.3 (How to Deal with Noise?)
You can rarely eliminate noise completely, but you can manage it.

(1) **Data Cleaning & Preprocessing**: This is the first line of defense.

- **Smoothing**: Applying algorithms to average out random fluctuations in time-series or signal data.
- **Binning**: Grouping numerical values into bins to smooth out small irregularities.
- **Correcting Typos**: Standardizing text entries.

(2) **Collecting More Data**: A larger dataset can sometimes "drown out" the noise, as the true signal becomes stronger with more examples.

(3) **Using Robust Algorithms**: Some machine learning models (like Random Forests) are inherently more resistant to noise than others (like Decision Trees).

(4) **Feature Selection**: Removing irrelevant or redundant features that mostly contain noise.

Idea 1.7.1
To determine whether our machine learning algorithm performs well not only on the training dataset but also generalizes well to new data, *we also want to randomly divide the dataset into separate training and test datasets*. In this way, we can measure whether our machine learning model is performing well.

1.7.2 TRAINING AND SELECTING A PREDICTIVE MODEL

We cannot get learning for free.

Observation 1.7.4
Meany different machine learning models have been developed to solve different problem tasks.

The goal is that sometimes is easier to use a certain machine learning model for some tasks than use it for other ones.

> **Example 1.7.2**
> Each clasification algorithm has its inherent biases, and no single calssification model enjoys superiority if we do not make any assumptions about the task.

In summary, it's essencial to compare at least a handful of different learning algotrithms in order to train and select the best performing model.

> **Observation 1.7.5**
> Before deciding which models we shall use, we have to decide upon a metric to measure performance. One commonly used metric is classification accuracy, which is defined as the proportion of correctly classified instances.

Sometimes we may have a machine learning model which was trained using a dataset for model selection, but one question arises: what if we do not use this training dataset? To adress this issue, we can use different techniques such as cross-validation.

> **Definition 1.7.3 (Cross-Validation)**
> **Cross-validation** *we devide further a dataset into training and validation subsets in order to estimate the generalization performance of the model.*

> **Observation 1.7.6 (Hyperparameters)**
> We also cannot expect that the default parameters of the different learning algorithms provided by software libraries are optimal for our specific problem task. Therefore, we will make frequent use of hyperparameter optimization techniques that help us fine-tune the performance of our model in later sections.
>
> We can think of those hyperparameters as parameters that are not learned from the data but represent the knobs of a model that we can turn to improve its performance.

1.7.3 Evaluating Models and Predicting Unseen Data Instances

After selecting a model that has been fitted on the training dataset, we can use the test dataset to estimate how well it performs on unseen data to determine the **generalization error**.

> **Definition 1.7.4 (Generalization Error)**
> **Generalization Error** is *formally the expected error of the model on a new, random instance drawn from the same underlying data distribution.*

> **Observation 1.7.7 (Dataset Dependancy)**
> It is important to note that the parameters for previously mentioned procedures, such as feature scaling and dimensionality reduction, **are solely obtained from the training dataset**, and the **same parameters are later reapplied to transform the test dataset as well as any new data instances—the performance measured on the test data may be overly optimistic otherwise**.

TRAINING SIMPLE MACHINE LEARNING ALGORITHMS FOR CLASSIFICATION

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

2.1 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 2.1.1**
> 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

2.2 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 2.2.1** (**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 2.2.1**
> They described a nerve cell as a **simple logic gate with binary outputs**.
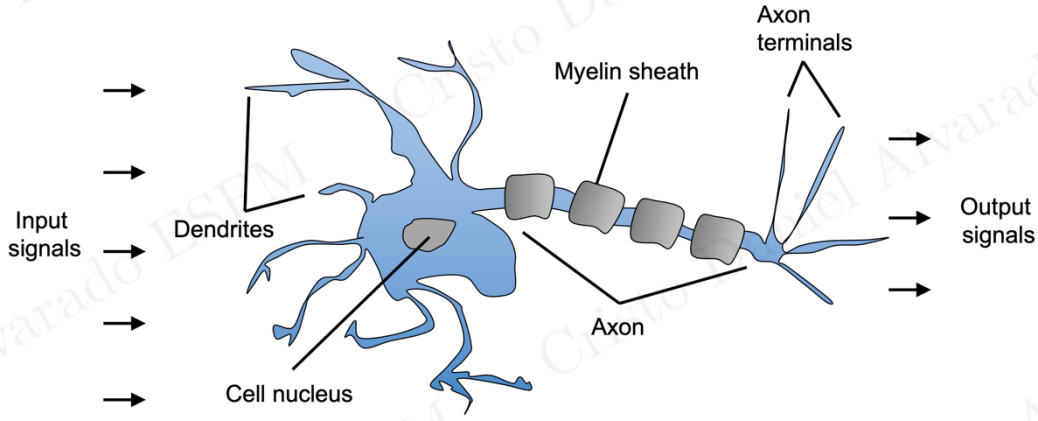
Figure 2.1: Biological Neuron.

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.

> **Observation 2.2.2** (**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.*

2.3   Formal Definition of an Artificial Neuron

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

> **Definition 2.3.1** (**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:
>
> $$z = w_1 x_1 + w_2 x_2 + \cdots + w_m x_m = \sum_{i=1}^{m} w_i x_i$$
>
> 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**:

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

17

Observation 2.3.1

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 2.3.2

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 2.3.3 (Notation Problem)

This notation is horrible and differs from the one Mathmaticians use in all his textbooks, please, don't use this notation.

2.4 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 2.4.1 (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,

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

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 2.4.3 (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 2.4.4
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 2.4.1
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.

2.5 IMPLEMENTING A PERCEPTRON LEARNING ALGORITHM IN PYTHON

Now, we will do an implementation of the Perceptron Rule in Python.

2.6 AND OBJECT-ORIENTED PERCEPTRON API

We will take an object-oriented approach to defining the perceptron interface a a Python class, wich will allow us to initialize new Perceptron objects that can learn from data via a `fit` method and make predictions via a separate `predict` method.

The code is the following:

```
1  import numpy as np
2
3  class Perceptron:
4      """Perceptron classifier.
5
6      Parameters
7      ------------
8      eta : float
9        Learning rate (between 0.0 and 1.0)
10     n_iter : int
11       Passes over the training dataset.
12     random_state : int
13       Random number generator seed for random weight
14       initialization.
15
16     Attributes
17     -----------
18     w_ : 1d-array
19       Weights after fitting.
20     b_ : Scalar
21       Bias unit after fitting.
22     errors_ : list
23       Number of misclassifications (updates) in each epoch.
24
25     """
26     def __init__(self, eta=0.01, n_iter=50, random_state=1):
27         self.eta = eta
28         self.n_iter = n_iter
29         self.random_state = random_state
30
31     def fit(self, X, y):
32         """Fit training data.
33
34         Parameters
35         ----------
36         X : {array-like}, shape = [n_examples, n_features]
37           Training vectors, where n_examples is the number of
38           examples and n_features is the number of features.
39         y : array-like, shape = [n_examples]
40           Target values.
41
42         Returns
43         -------
44         self : object
45
46         """
```

```
47          rgen = np.random.RandomState(self.random_state)
48          self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape
               [1])
49          self.b_ = np.float_(0.)
50          self.errors_ = []
51
52          for _ in range(self.n_iter):
53              errors = 0
54              for xi, target in zip(X, y):
55                  update = self.eta * (target - self.predict(xi))
56                  self.w_ += update * xi
57                  self.b_ += update
58                  errors += int(update != 0.0)
59              self.errors_.append(errors)
60          return self
61
62      def net_input(self, X):
63          """Calculate net input"""
64          return np.dot(X, self.w_) + self.b_
65
66      def predict(self, X):
67          """Return class label after unit step"""
68          return np.where(self.net_input(X) >= 0.0, 1, 0)
```

Code 2.1: Python Perceptron Rule Implementation.

First, we will list the parameters and attributes of the code, in order to understand it better:

- eta: Is a float, which represents the learning rate (a number between 0.0 and 1.0).

- n_iter: Is an integer, which is the number of passes over the training dataset.

- random_state: It is an integer, which is a **random number generator seed** for random weight initialization.

Definition 2.6.1 (Random Number Generator (RNG))
A **random number generator (RNG) seed** is *an initial value used by a computer algorithm to generate a sequence of pseudo-random numbers.* For random weight *initialization in a neural network, this seed sets the starting point for the sequence of numbers used to initialize the model's weights and biases.*

The attributes are the following:

- w_: It is a 1-dimensional array, which represents the weights after fitting.

- b_: It is a float, which represents the bias unit after fitting.

- errors_: It is a list, which is the number of misclassifications (or updates) in each **epoch**.

Definition 2.6.2 (Epoch)
In machine learning, an **epoch** *is one complete pass through the entire training dataset.* During a single epoch, *the learning algorithm processes every training example once to update the model's internal parameters.* Training a model over multiple epochs *allows it to progressively learn and*

> *refine its understanding of the data, with a well-tuned number of epochs balancing performance between underfitting and overfitting.*

In this code, we have three methods in the class Perceptron:

- `__init__`: It initializes the parameters of the class, in this case, the `eta` is setted to `0.01`, `n_iter`=50 and `random_state`=1.

- `fit`: It is the code that fits the data in a dataset, given an array `X` and another array `y` of target values.

- `net_input`: It calculates the net input of a dataset `X`.

- `predict`: It calculates the class label after a unit step of a dataset `X`.

Observation 2.6.2 (**Use of the Perceptron Implementation**)
Using this perceptron implementation, we can initialize new Perceptron objects with a given learning rate, `eta`, and the number of epochs, `n_iter` (or iterations over the training dataset).

Via the `fit` method, we initialize the bias `self.b_` to an initial value `0` and the weights in `self.w_` to a vector in $\mathbb{R}^m$. Here, m stands for the number of dimensions (features) in the dataset.

Observation 2.6.3 (**Decision of Initialization of Weight Vectors**)
The **initial weight** vector contains small random numbers drawn from a normal distribution with a standard deviation of `0.01` via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape [1])`, where `rgen` is a NumPy random number generator seeded with a user-specified random seed so that results can be reproduced when desired.

Technically, *we could initialize the weights to zero (this is done in the original perceptron algorithm).* However, if we did that, the **learning rate** η (`eta`) *would have no effect on the decision boundary. If all the weights are initialized to zero, the learning rate parameter affects only the scale of the weight vector, not the direction.*

Observation 2.6.4 (**Use of Normal Distribution**)
Our decision to draw the random numbers from a normal distribution—for example, instead of from a uniform distribution—and to use a standard deviation of `0.01` was arbitrary; remember, we are just interested in small random values to avoid the properties of all-zero vectors, as discussed earlier.

Excercise 2.6.1
As an optional exercise, you can change `self.w_ = rgen.normal(loc=0.0, scale=0.01, size =X.shape[1])` to `self.w_ = np.zeros(X.shape[1])` and *run the perceptron training code presented in the next section with different values for* `eta`. *You will observe that the decision boundary does not change.*

After the weights have been initialized, the fit method loops over all individual examples in the training dataset and updates the weights according to the perceptron learning rule discussed in the previous section.

Observation 2.6.5 (**Prediction**)
The **class labels are predicted by the predict method**, which is *called in the fit method during training to get the class label for the weight update*; but *predict can also be used to predict the class labels of new data after the model has been fitted*. Furthermore, we collect the number of misclassifications during each epoch in the `self.errors_` list so that *we can later analyze how well the perceptron performed during training*. The `np.dot` function used in the `net_input` method calculates the vector dot product, $wTx + bwTx + b$.

Idea 2.6.1
Instead of using `NumPy` to calculate the vector dot product between two arrays, `a` and `b`, via `a.dot(b)` or `np.dot(a, b)`, we could also *perform the calculation in pure Python* via `sum([i * j for i, j in zip(a, b)])`. However, the advantage of using `NumPy` over classic Python for loop structures is that **its arithmetic operations are vectorized**. *Vectorization means that an elemental arithmetic operation is automatically applied to all elements in an array.* By formulating our arithmetic operations as a *sequence of instructions on an array, rather than performing a set of operations for each element at a time, we can make better use of our modern central processing unit* (**CPU**) architectures with single instruction, multiple data (**SIMD**) support. Furthermore, `NumPy` *uses highly optimized linear algebra libraries, such as* **Basic Linear Algebra Subprograms** (**BLAS**) and **Linear Algebra Package** (**LAPACK**), that have been written in `C` or `Fortran`. Lastly, `NumPy` also *allows us to write our code in a more compact and intuitive way using the basics of linear algebra, such as vector and matrix dot products.*

2.7 TRAINING OF THE PERCEPTRON ALGOTITHM ON THE IRIS DATASET

Observation 2.7.1 (**Restriction of Dimensions**)
To *test our perceptron implementation, we will restrict the following analyses and examples in the remainder of this section to two feature variables* (dimensions). Although the perceptron rule is not restricted to two dimensions, *considering only two features—sepal length and petal length—allows us to visualize the decision regions of the trained model in a scatterplot.*

Idea 2.7.1 (**Restriction to Only Two Classes**)
We will also only consider two flower classes, setosa and versicolor, from the Iris dataset for practical reasons (remember, the **perceptron is a binary classifier**). However, the *perceptron algorithm can be extended to multi-class classification using the one-versus-all (OvA) technique.*

2.7.1 OVA METHOD FOR MULTI-CLASS CLASSIFICATION

We can extend any binary classifier to a multi-class problems using the method called **OvA** or **one-versus-rest** (**OvR**).
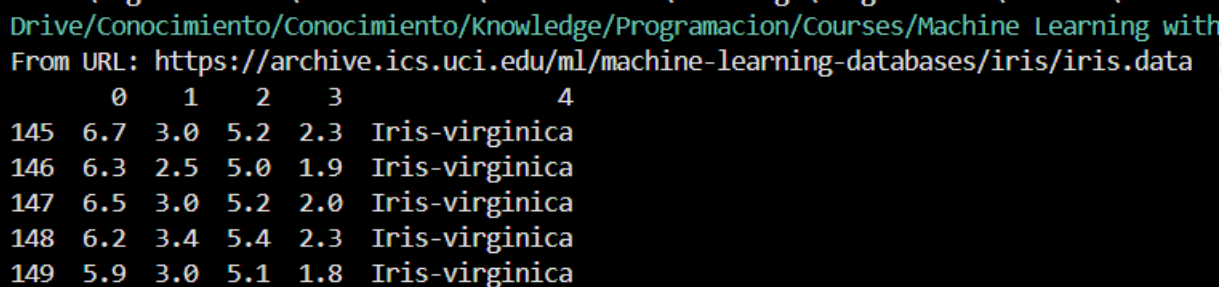
Definition 2.7.1 (**One-versus-rest (OvR)**)
The method **OvR** allows to extend any binary classifier to a multple classifier. To classify a new, unlabeled instance, we use our n classifiers (where n is the number of class labels) and *assign the class label with the highest confidence to the instance.* For the perceptron, **we choose the class label associated with the largest absolute net input value**.

First, we use the `pandas` library to load the Iris dataset directly from the **UCI Machine Learning Repository** into a `DataFrame` **object** and *print the last five lines via the tail method to confirm that the data loaded correctly*:

```
1   import os
2   import pandas as pd
3   s = 'https://archive.ics.uci.edu/ml/'\
4       'machine-learning-databases/iris/iris.data'
5   print('From URL:', s)
6   #From URL: https://archive.ics.uci.edu/ml/machine-learning-
        databases/iris/iris.data
7   df = pd.read_csv(s,
8                    header=None,
9                    encoding='utf-8')
10  print(df.tail())
```

Code 2.2: Read Dataset from UCI.

Figure 2.2: Tail of the Iris Dataset.

Next, *we extract the first* 100 *class labels that correspond to the* 50 `Iris-setosa` and 50 `Iris-versicolor` *flowers and convert the class labels into the two integer class labels* 1 (`versicolor`) *and* 0 (`setosa`), *assigning them to a vector* `y`. Similarly, *we extract the first feature column* (`sepal length`) *and the third feature column* (`petal length`) *of those* 100 *training examples and assign them to a feature matrix* `X`, *which we then visualize via a two-dimensional scatterplot*:

```
1   import matplotlib.pyplot as plt
2   import numpy as np
3   # select setosa and versicolor
4   y = df.iloc[0:100, 4].values
5   y = np.where(y == 'Iris-setosa', 0, 1)
6   # extract sepal length and petal length
7   X = df.iloc[0:100, [0, 2]].values
8   # plot data
9   plt.scatter(X[:50, 0], X[:50, 1],
10              color='red', marker='o', label='Setosa')
11  plt.scatter(X[50:100, 0], X[50:100, 1],
12              color='blue', marker='s', label='Versicolor')
13  plt.xlabel('Sepal length [cm]')
14  plt.ylabel('Petal length [cm]')
15  plt.legend(loc='upper left')
16  plt.show()
```

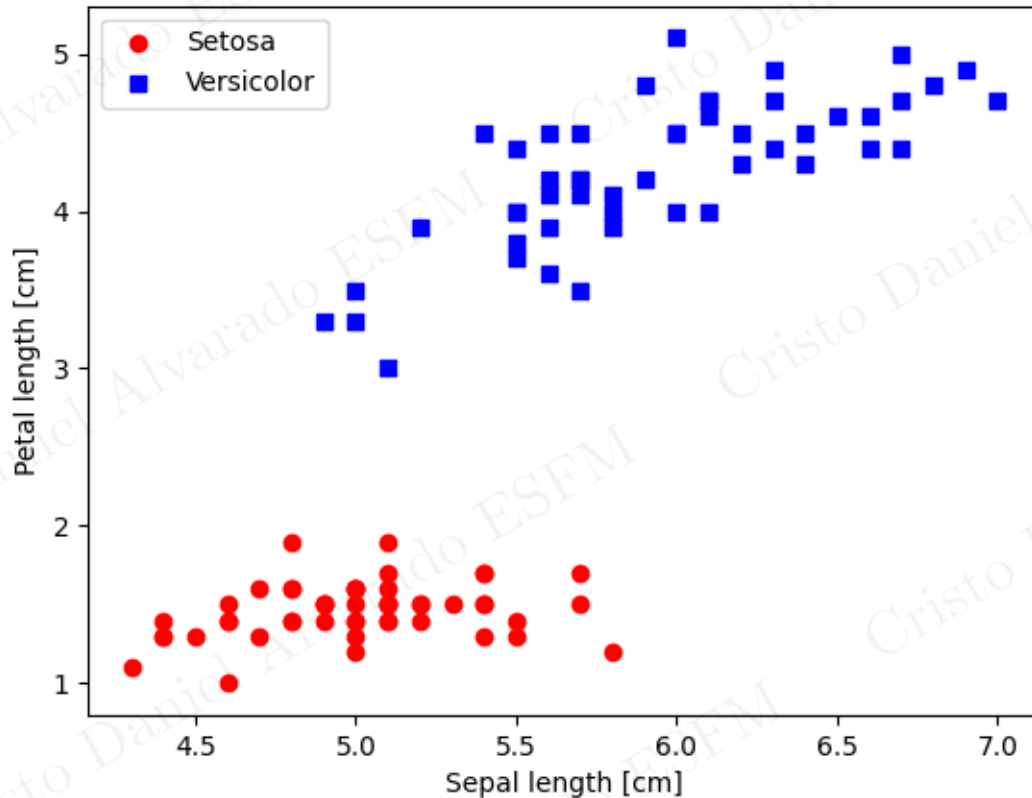Code 2.3: Extraction and Visualization of Iris Dataset.

Figure 2.3: Plot of the Seritosa and Versicolor on the Iris Dataset.

This plot shows the distribution of flower examples in the Iris dataset, along with the two feature axes: petal and sepal length.

Observation 2.7.3
In this subspace, a linear decision boundary should be sufficient to separate setosa from versicolor flowers.

Thus, a linear classifier such as the perceptron should be able to classify the flowers in the Iris dataset perfectly.

2.7.2 TRAINING THE ALGORITHM

We will also plot the misclassification error for each epoch to check whether the algorithm converged and found a decision boundary that separates the two Iris flower classes:

```
import Perceptron
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)
plt.plot(range(1, len(ppn.errors_) + 1),
         ppn.errors_, marker='o')
plt.xlabel('Epochs')
```

```
7  plt.ylabel('Number of updates')
8  plt.show()
```

Code 2.4: Training the Perceptron Algorithm with the Iris Dataset
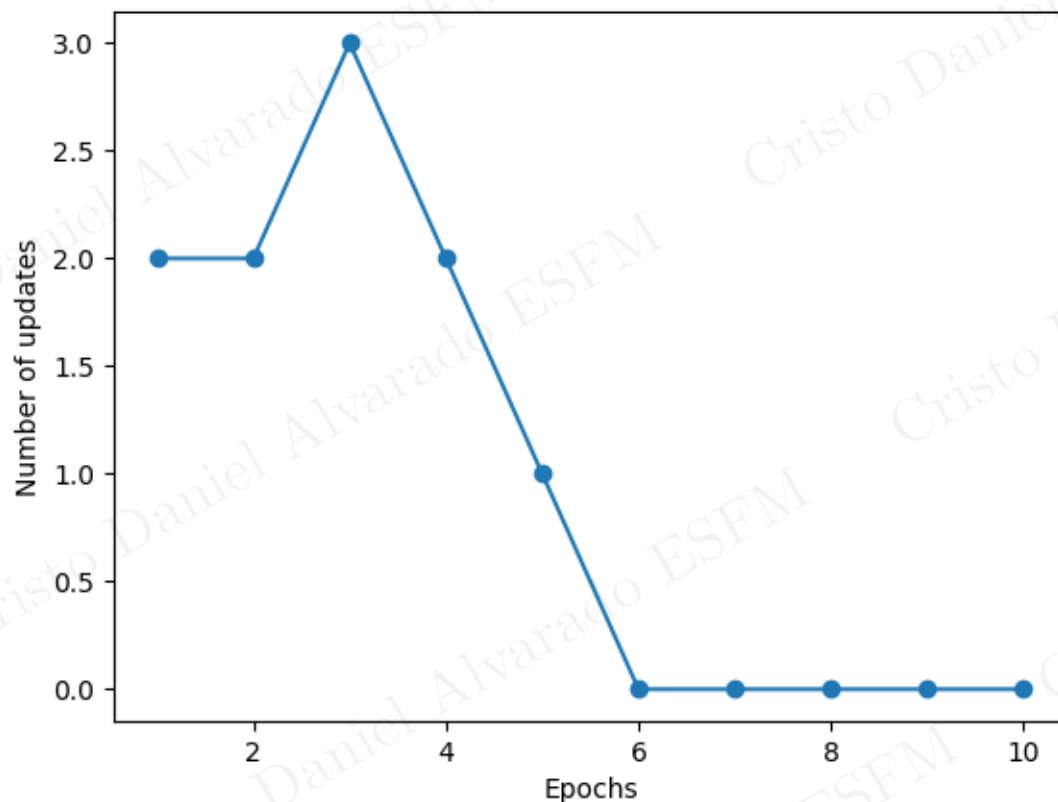
Figure 2.4: Epoch Plot and Number of Updates on the Iris Dataset.

Let's *implement a small convenience function to visualize the decision boundaries for two-dimensional datasets*:

```
1  from matplotlib.colors import ListedColormap
2
3  def plot_decision_regions(X, y, classifier, resolution=0.02):
4      # setup marker generator and color map
5      markers = ('o', 's', '^', 'v', '<')
6      colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
7      cmap = ListedColormap(colors[:len(np.unique(y))])
```

27

```
8
9       # plot the decision surface
10      x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
11      x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
12      xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution)
           ,
13                              np.arange(x2_min, x2_max, resolution)
                                  )
14      lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()
           ]).T)
15      lab = lab.reshape(xx1.shape)
16      plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
17      plt.xlim(xx1.min(), xx1.max())
18      plt.ylim(xx2.min(), xx2.max())
19
20      # plot class examples
21      for idx, cl in enumerate(np.unique(y)):
22          plt.scatter(x=X[y == cl, 0],
23                      y=X[y == cl, 1],
24                      alpha=0.8,
25                      c=colors[idx],
26                      marker=markers[idx],
27                      label=f'Class {cl}',
28                      edgecolor='black')
```

Code 2.5: Plotting Regions on the Iris Dataset

First, we *define a number of* `colors` *and* `markers` *and create a colormap from the list of colors* via `ListedColormap`. Then, we *determine the minimum and maximum values for the two features and use those feature vectors to create a pair of grid arrays,* `xx1` *and* `xx2`, *via NumPy's meshgrid function.* Since we trained our perceptron classifier on two feature dimensions, *we flatten the grid arrays and create a matrix with the same number of columns as the Iris training subset so that we can use the predict method to predict the class labels lab of the corresponding grid points.*

After *reshaping the predicted class labels into a grid with the same dimensions as* `xx1` *and* `xx2`, *we draw a contour plot via Matplotlib's contourf function, which maps the different decision regions to different colors for each predicted class in the grid array*:

```
1  plot_decision_regions(X, y, classifier=ppn)
2  plt.xlabel('Sepal length [cm]')
3  plt.ylabel('Petal length [cm]')
4  plt.legend(loc='upper left')
5  plt.show()
```

Code 2.6: Plotting the Perceptron Info on the Iris Dataset

As we can see in the plot, the perceptron learned a decision boundary that can classify all flower examples in the Iris training subset perfectly.

Observation 2.7.6 (**Convergence of the Perceptron**)
Although the *perceptron classified the two Iris flower classes perfectly, convergence is one of its biggest challenges. Rosenblatt proved mathematically that* the **perceptron learning rule converges if the two classes can be separated by a linear hyperplane**. However, *if the*
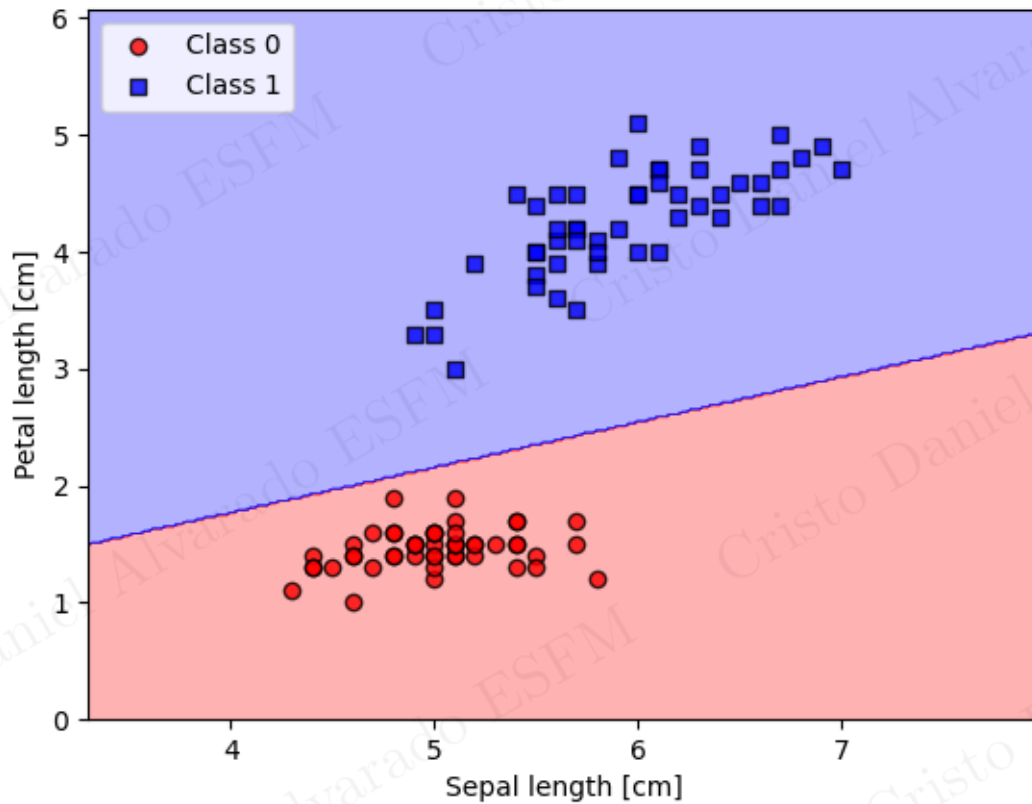
Figure 2.5: Perceptron Visual Classification on the Iris Dataset.

classes cannot be separated perfectly by such a linear decision boundary, the weights will never stop updating unless we set a maximum number of epochs. Interested readers can find a summary of the proof in the lecture notes at Perceptron Slides.

Appendix A

Using Python for Machine Learning

A.1 Basic Configuration

For machine learning tasks, we will mostly refer to the *Scikit-Learn* library, which is one of the most popular and accessible open-source machine learning libraries for python.

When we focus on a subfield of machine learning called: **deep learning**, we wiell use the latest version of the *PyTorch library*, which specializes in the training of the so called deep **netural network models**.

> **Definition A.1.1 (Scikit-learn)**
> **Scikit-learn** is a *classical machine learning library for Python. It is built on top of NumPy and SciPy and provides a clean, uniform, and simple API for a wide variety of traditional ML algorithms.*
>
> **PyTorch** is an *open-source deep learning framework developed primarily by Facebook's AI Research lab (FAIR). It provides the foundational building blocks for building and training neural networks, with a strong focus on flexibility and speed.*

We'll use version 3.9 of Python. To check the version of python we use the following code:

```
1  python --version
```
Code A.1: Check Python Version

> **Observation A.1.1 (Use of pip)**
> To install aditional packages used thorught the course, we can install them via the `pip` program.

> **Definition A.1.2 (Pip)**
> `pip` is the *standard package installer for Python. It is a command-line utility that allows users to install, manage, and uninstall Python packages and libraries that are not part of the Python standard library.*

After installing python, it's possible to execute the following comand in order to install some package:

```
1  pip install SomePackage
```
Code A.2: `pip` Package Installation.

A.2   Anaconda Python Distribution and Package Manager

> **Idea A.2.1**
>
> A recommended open-source package management system for installing python for scientific computing contexts is conda by Continuum Analytics.

> **Definition A.2.1 (Conda)**
>
> **Conda** is a *free and licensed under a permissive open-source licence.* Its goal is to help with the *installation and version management of Python packages for data science, math and engineering across different operating systems.*

Conda comes in different flavours, like a Linux installation. Some of them are **Anaconda**, **Miniconda** and **Minforge**.

- **Anaconda** comes with many scientific computing packages pre-installed. The Anaconda installer can be downloaded here and an Anaconda quick start guide is available here.

- **Miniconda** is a leaner alternative to Anaconda (here). Essentially, it is similar to Anaconda but without any packages pre-installed, which many people (including the authors) prefer.

- **Miniforge** is similar to Miniconda but community-maintained and uses a different package repository (conda-forge) from Miniconda and Anaconda. We found that Miniforge is a great alternative to Miniconda. Download and installation instructions can be found in the GitHub repository here.

After successfully installing conda through either Anaconda, Miniconda, or Miniforge, we can install and update, respectively, new Python packages using the following command:

```
1  conda install SomePackage
2  conda update SomePackage
```
Code A.3: Conda Package Installation.

Packages that are not available through the official conda channel might be available via the community-supported conda-forge project (conda-forge), which can be specified via the –channel conda-forge flag. For example:

```
1  conda install SomePackage --channel conda-forge
```
Code A.4: caption

Packages not available throught the default conda channel or conda-forge can be installed via pip as explainded earlier in Code A.2.

A.3   Packages for Scientific Computing, Data Science, and Machine Learning

We will use mainly NumPy's arrays and sometimes Pandas (highler level data manipulation tools). Also the Matplotlib library will be useful to visualize quantitative data. The version of the packages to install are the following:

- NumPy 1.21.2

- SciPy 1.7.0

- Scikit-learn 1.0

- Matplotlib 3.4.3

- Pandas 1.3.2

After installing these packages, you can double-check the installed version by importing the package in Python and accessing its `__version__` attribute:

```
1  > Use 'numpy.__version__' to check the installed version.
2  '1.21.2'
```

Code A.5: Check Version of Packages