Machine Learning with PyTorch and Scikit-Learn From Coursera

Cristo Daniel Alvarado

November 11, 2025

Contents

1 Module 1 **4**

1.1 Overview . . . 4

1.2 Giving computers the ability to learn from data . . . 4

1.3 Introduction . . . 4

1.3.1 The Three Different Types of Machine Learning . . . 5

1.3.2 Supervised Learning . . . 5

1.3.3 Classification for Predicting Class Labels . . . 6

1.3.4 Regression for Predicting Continous Outcomes . . . 7

1.4 Solving Problems with Reinforcement Learning . . . 8

1.5 Unsupervised Learning . . . 9

1.5.1 Finding Subgroups With Clustering . . . 9

1.5.2 Dimensionality Reduction for Data Compresion . . . 9

1.6 Terminology and Notations . . . 10

1.6.1 Machine Learning Teminology . . . 12

1.7 Machine Learning Systems . . . 12

1.7.1 Preprocessing: Getting Data into Shape . . . 12

1.7.2 Training and Selecting a Predictive Model . . . 14

1.7.3 Evaluating Models and Predicting Unseen Data Instances . . . 15

2 Training Simple Machine Learning Algorithms for Classification **16**

2.1 Introduction . . . 16

2.2 Artificial Neurons: Early Stages of Machine Learning . . . 16

2.3 Formal Definition of an Artificial Neuron . . . 17

2.4 Perceptron Learning Rule . . . 18

2.5 Implementing a Perceptron Learning Algorithm in Python . . . 20

2.6 And Object-Oriented Perceptron API . . . 20

2.7 Training of the Perceptron Algotithm on the Iris Dataset . . . 24

2.7.1 OvA method for multi-class classification . . . 24

2.7.2 Training the Algorithm . . . 26

2.8 Adaptive Linear Neurons and the Convergence of Learning . . . 30

2.8.1 Main Differenes Between Perceptron and Adaline . . . 30

2.8.2 Minimizing Loss Function with Gradient Descent . . . 30
2.9 Implementing Adaline in Python . . . 33
2.9.1 Hyperparameters . . . 36
2.10 Improving Gradient Descent Through Feature Scaling . . . 36
2.10.1 Large-Scale Machine Learning and Stochastic Gradient Descent . . . 38

3 A Tour of Machine Learning Classifiers Using Scikit-Learn 43
3.1 Introduction . . . 43
3.1.1 Choosing a Classification Algorithm . . . 43
3.1.2 First Steps with scikit-learn Training a Perceptron . . . 44
3.1.3 Training a Model Using Scikit-Learn . . . 44
3.2 Modeling Class Probabilities Via Logistic Regression . . . 49
3.2.1 Logistic Regression and Conditional Probabilities . . . 49
3.3 Learning the Model Weights via the Logistic Loss Function . . . 52
3.3.1 Deriving the likelihood function . . . 52
3.4 Converting an Adaline Implementation Into an Algorithm for Logistic Regression . . . 53
3.5 Gradient Descent . . . 56
3.6 Training a Logistic Regression Model with Scikit-Learn . . . 56
3.6.1 Algorithms for convex optimization . . . 58
3.7 Tackling Overfitting via Regularization . . . 60

A Using Python for Machine Learning 61
A.1 Basic Configuration . . . 61
A.2 Anaconda Python Distribution and Package Manager . . . 62
A.3 Packages for Scientific Computing, Data Science, and Machine Learning . . . 62

B A Note on Distributions 64
B.1 Normal Distrubution . . . 64
B.2 Bernoulli's Distribution . . . 64

Code List

2.1 Python Perceptron Rule Implementation. 21
2.2 Read Dataset from UCI. 25
2.3 Extraction and Visualization of Iris Dataset. 25
2.4 Training the Perceptron Algorithm with the Iris Dataset 26
2.5 Plotting Regions on the Iris Dataset 27
2.6 Plotting the Perceptron Info on the Iris Dataset 28
2.7 Adaline Implementation in Python. 34
2.8 Computation of Partial Derivatives 35
2.9 Weight Update Optimization. 35
2.10 Loss for Two Different Learning Rates. 36
2.11 Standarization Using Numpy 38
2.12 Adaline With Standarization Procedure 38
2.13 Adaline Implemented Using Stochastic Gradient Descent 40
3.1 Class Labels of Matrix `X` 44
3.2 Train Test Split Function 45
3.3 Use of `bincount` Method in Python. 45
3.4 `StandardScaler` Method in Python. 45
3.5 Perceptron Trained Model using Sklearn 46
3.6 Sigmoid Function Graph. 50
3.7 Implementation of Logistic Regression in Python. 54
3.8 Logistic Regression Iris Dataset Python. 57
3.9 Predicting Probabilities. 59
A.1 Check Python Version 61
A.2 `pip` Package Installation. 61
A.3 Conda Package Installation. 62
A.4 caption 62
A.5 Check Version of Packages 63

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.*

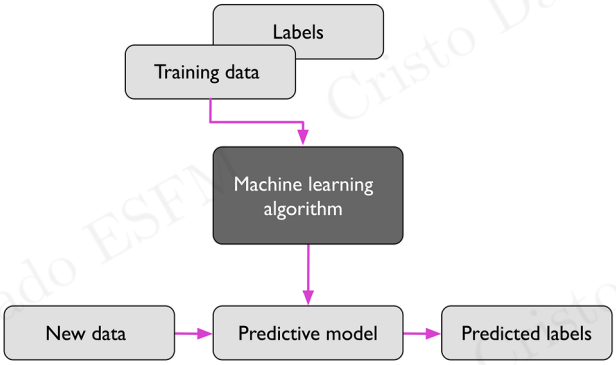

Figure 1.1: General Process of Supervised Learning

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.

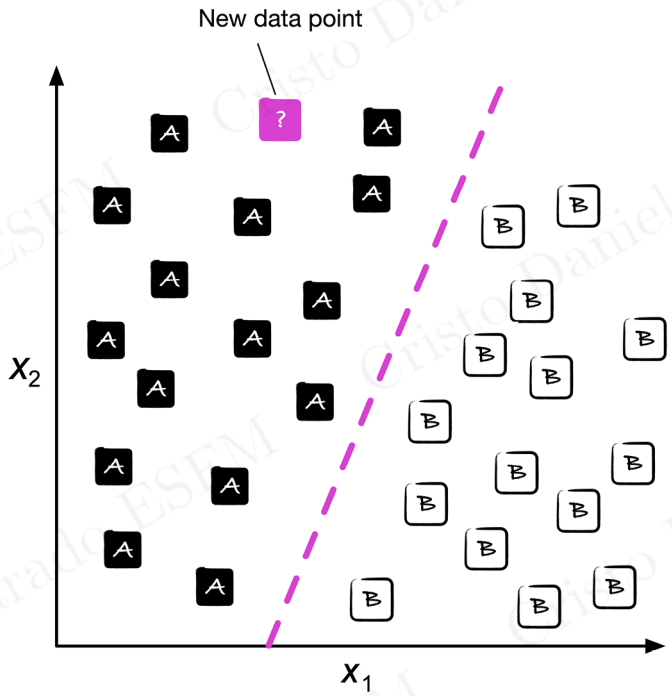

Figure 1.2: Classification in Machine Learning

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.

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.

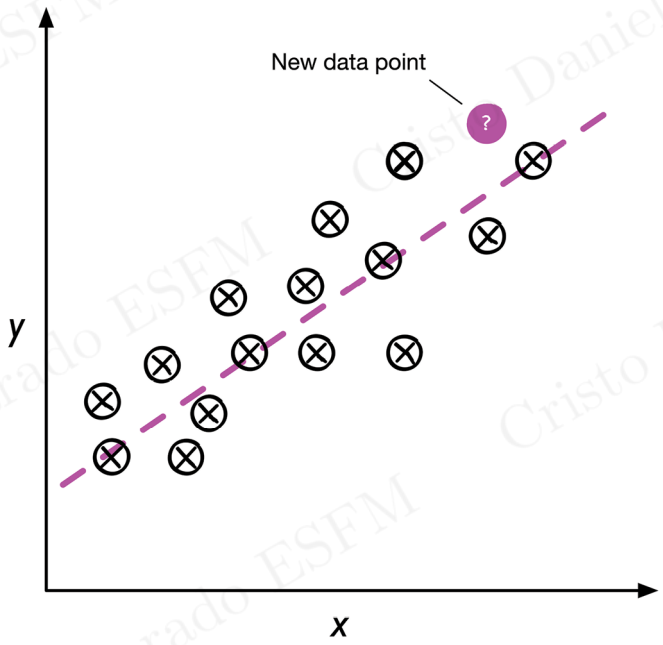

Figure 1.3: Linear Aproximation

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.

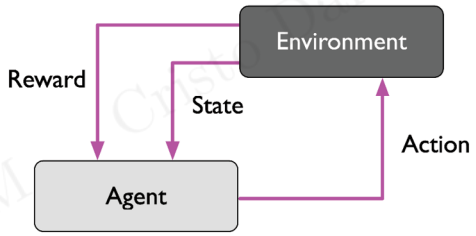

Figure 1.4: Reinforcement Learning Process

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.

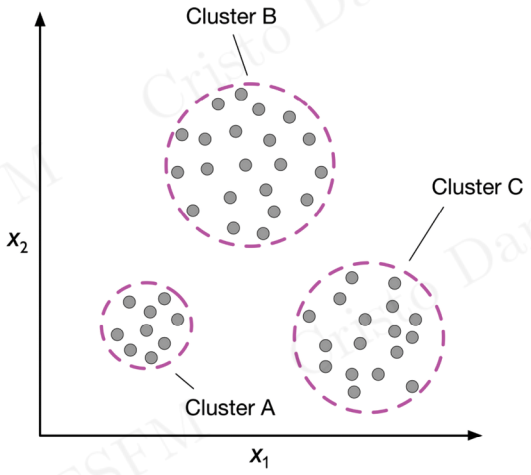

Figure 1.5: Caption

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.

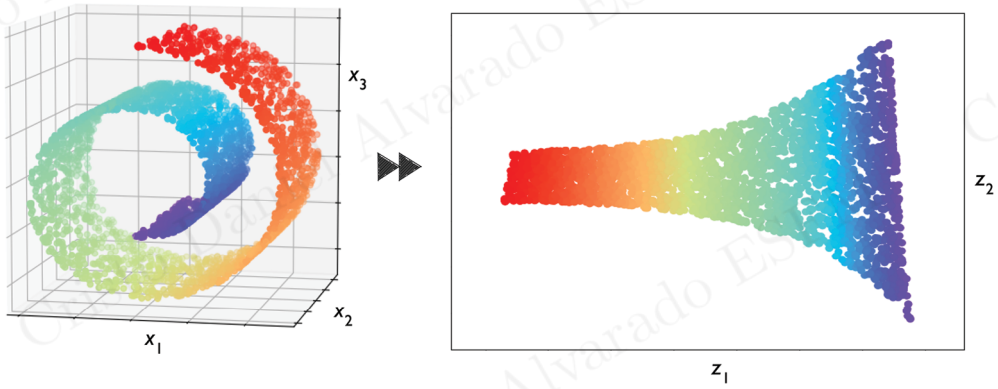

Figure 1.6: Example of Dimensionality Reduction in a Dataset.

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:

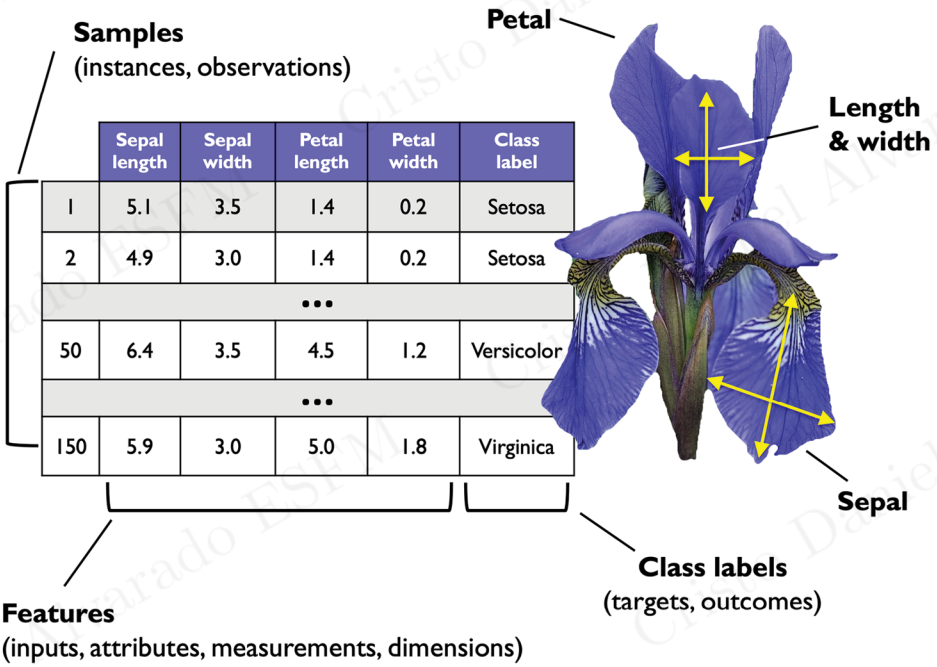

Figure 1.7: Samples, Features, and Class labels.

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:

$$x^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$$

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

$$x_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}$$

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

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(150)} \end{bmatrix}$$

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.

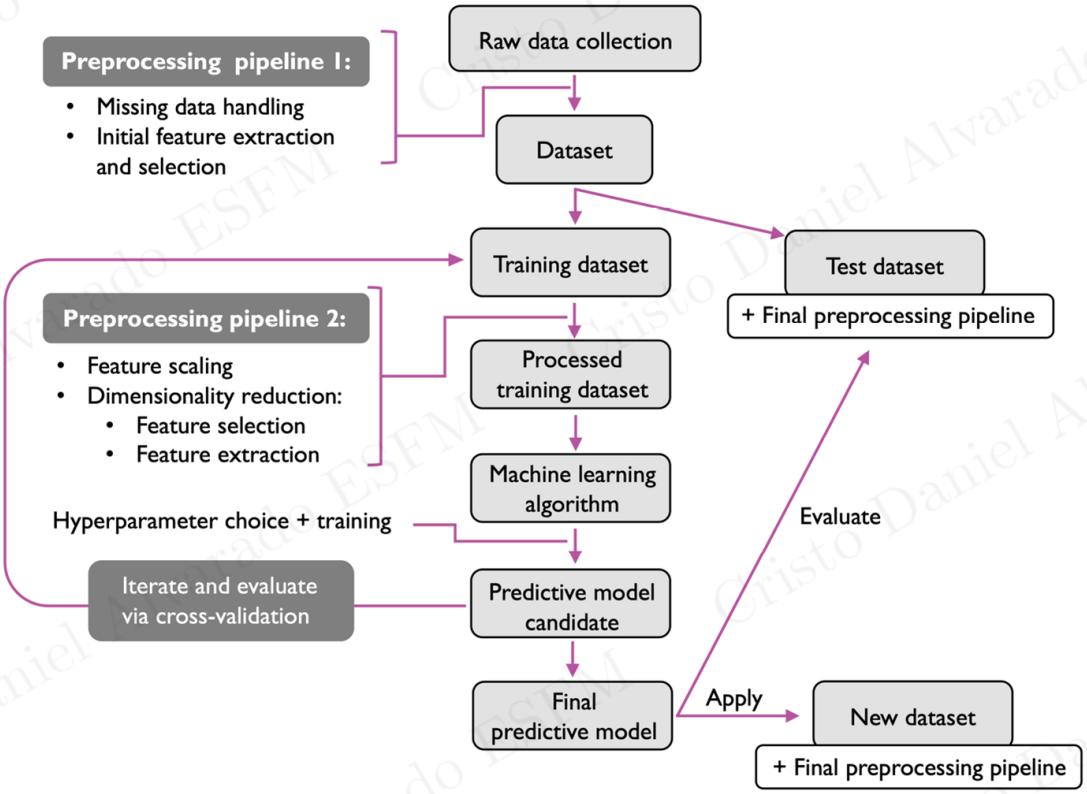

Figure 1.8: Machine Learning System.

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**.

Chapter 2

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**.

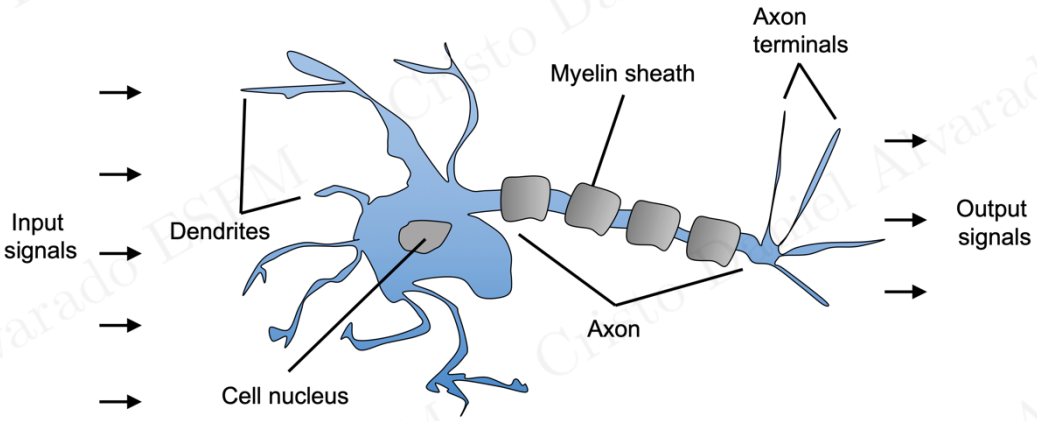

Figure 2.1: Biological Neuron.

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 **articial 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**:

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

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:

$$z' = w_1x_1 + w_2x_2 + \cdots + w_mx_m + b = \vec{w}^T\vec{x} + b$$

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,

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

Observation 2.4.2 (**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 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.

Observation 2.6.1
We append an underscore _ to attributes that are not created upon the inicialization of the object, but we do this by calling the object's other methods, for example, `self._w_`.

The code is the following:

```
import numpy as np

class Perceptron:
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    random_state : int
      Random number generator seed for random weight
      initialization.

    Attributes
    -----------
    w_ : 1d-array
      Weights after fitting.
    b_ : Scalar
      Bias unit after fitting.
    errors_ : list
      Number of misclassifications (updates) in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
          Training vectors, where n_examples is the number of
          examples and n_features is the number of features.
        y : array-like, shape = [n_examples]
          Target values.

        Returns
        -------
        self : object

        """
```

```
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape
            [1])
        self.b_ = np.float_(0.)
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_ += update * xi
                self.b_ += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        """Return class label after unit step"""
        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*:

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

Code 2.2: Read Dataset from UCI.

```
Drive/Conocimiento/Conocimiento/Knowledge/Programacion/Courses/Machine Learning with
From URL: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
     0    1    2    3               4
145  6.7  3.0  5.2  2.3  Iris-virginica
146  6.3  2.5  5.0  1.9  Iris-virginica
147  6.5  3.0  5.2  2.0  Iris-virginica
148  6.2  3.4  5.4  2.3  Iris-virginica
149  5.9  3.0  5.1  1.8  Iris-virginica
```

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*:

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

Code 2.3: Extraction and Visualization of Iris Dataset.

Observation 2.7.2 (**Use of the Code**)
Add this code to the remaining code.

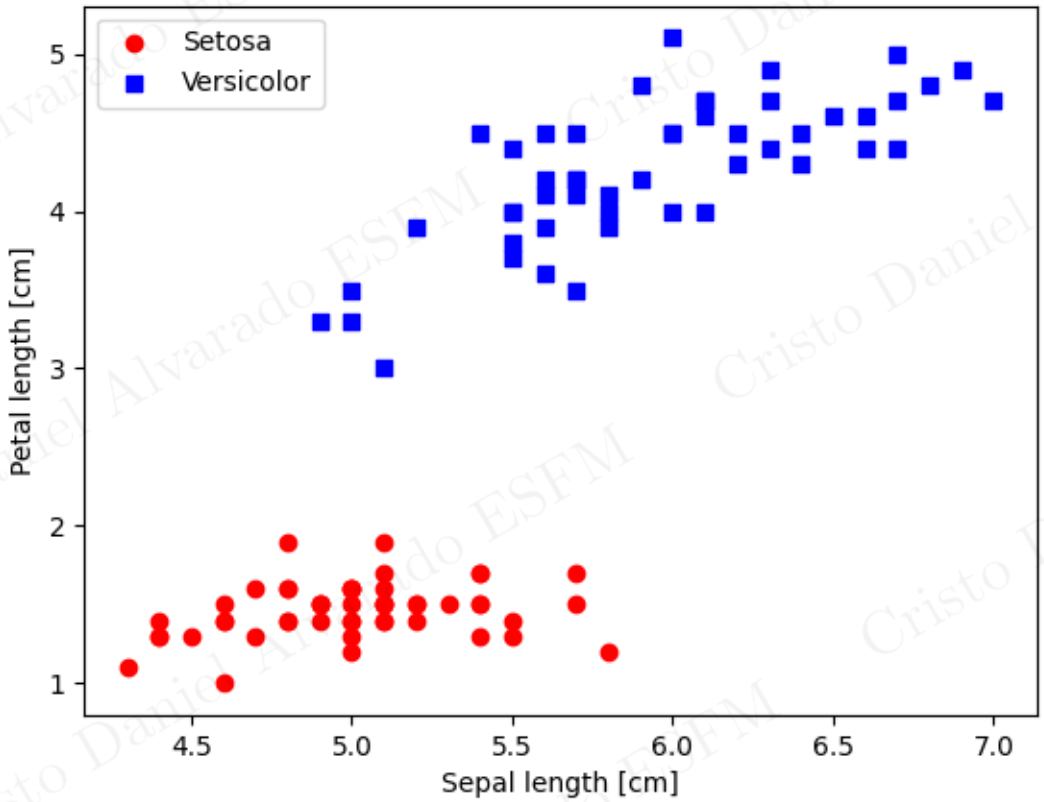

Figure 2.3: Plot of the Seritosa and Versicolor on the Iris Dataset.

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')
```

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

Code 2.4: Training the Perceptron Algorithm with the Iris Dataset

Observation 2.7.4
In this scenario, since I put Perceptron in a different Python file, I have to import it in order to be able to use it.

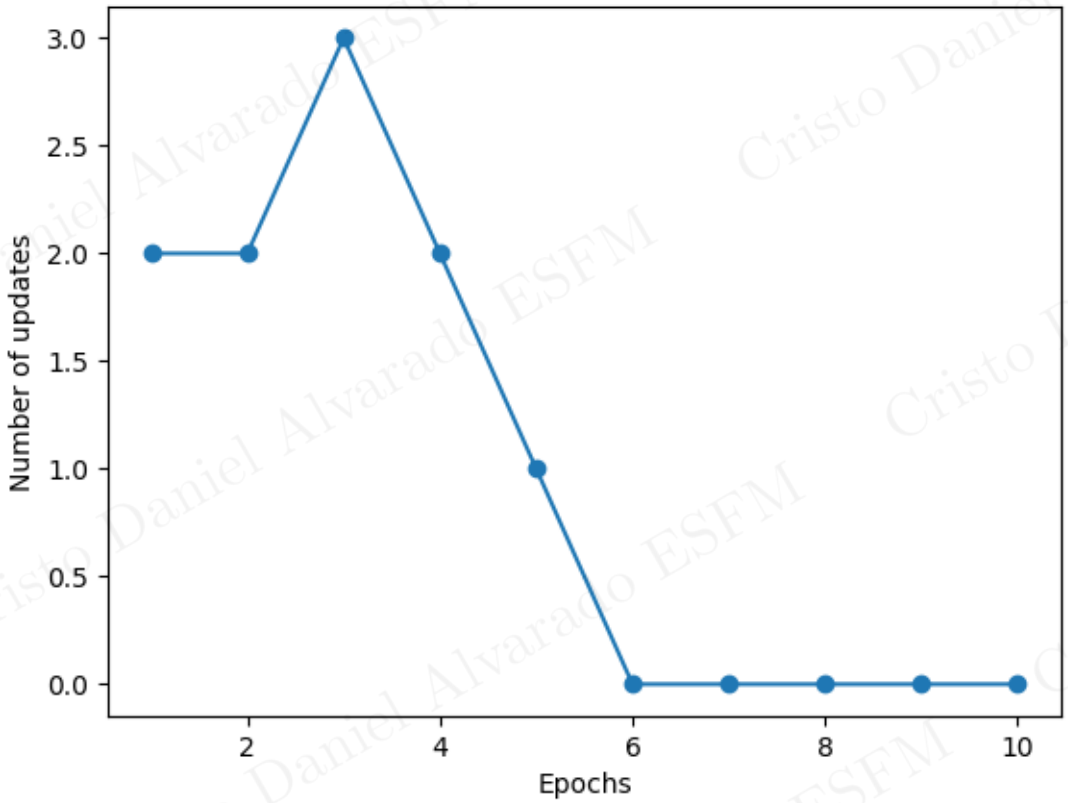

Figure 2.4: Epoch Plot and Number of Updates on the Iris Dataset.

Observation 2.7.5 (Convergence of the Perceptron)
As we can see, our **perceptron converged after the sixth epoch and should now be able to classify the training examples perfectly**.

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

```
from matplotlib.colors import ListedColormap

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

```
    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class examples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    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*:

```
plot_decision_regions(X, y, classifier=ppn)
plt.xlabel('Sepal length [cm]')
plt.ylabel('Petal length [cm]')
plt.legend(loc='upper left')
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*

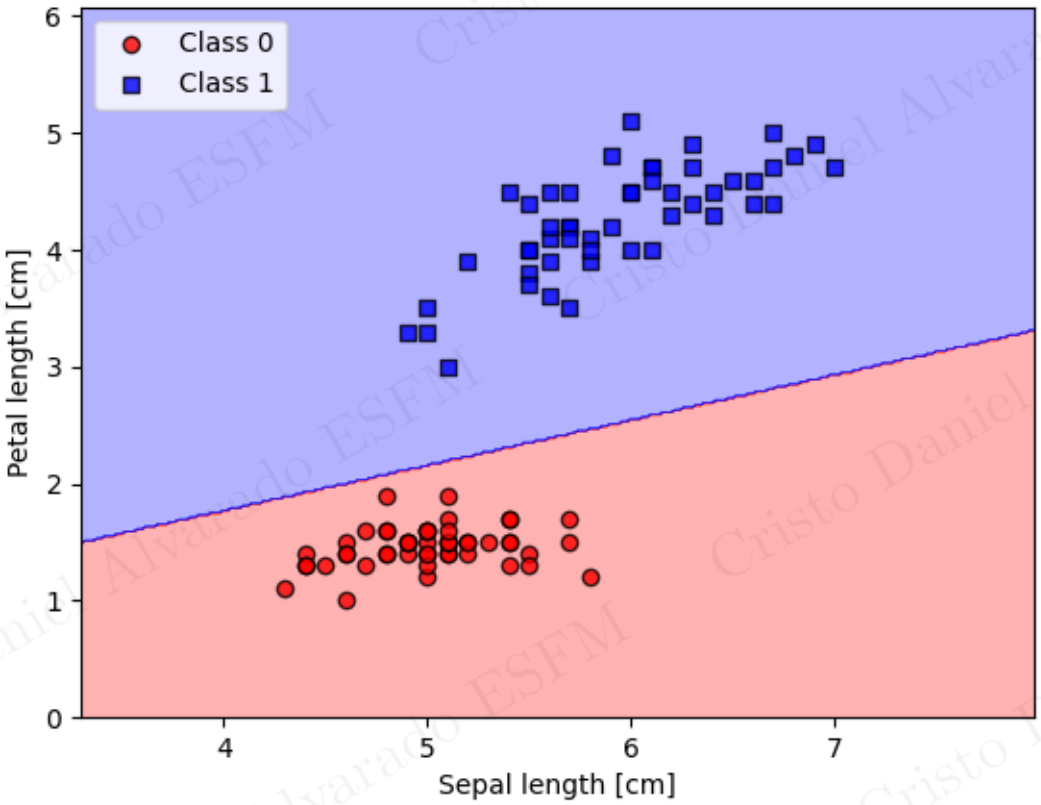

Figure 2.5: Perceptron Visual Classification on the Iris Dataset.

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.

2.8 Adaptive Linear Neurons and the Convergence of Learning

ADAptive LInear NEuron (**Adaline**). **Adaline** was *published by Bernard Widrow and his doctoral student Tedd Hoff only a few years after Rosenblatt's perceptron algorithm*, and it can be considered *an improvement on the latter* (An Adaptive "Adaline" Neuron Using Chemical "Memistors", Technical Report Number 1553-2 by B. Widrow and colleagues, Stanford Electron Labs, Stanford, CA, October 1960).

This algorithm is interesting because it ilustrates the key concepts of defining and minimizing a continuous loss function.

Observation 2.8.1 (**Use of Adaline**)
This *lays the groundwork for understanding other machine learning algorithms for classification*, such as logistic regression, support vector machines, and multilayer neural networks, as well as linear regression models.

Idea 2.8.1 (**Widrow-Hoff Rule**)
The key difference between Adaline and the Perceptron is that weights are updated based on a linear activation function rather than a unit step function.

This is called **Widrow-Hoff rule**.

In Adaline, the linear activation function is the identity function of the net input, that is: $\sigma(z) = z$.

Observation 2.8.2
While the linear activation function is used for learning the weights, a threshold function is still applied to make the final prediction, which is similar to the unit step function covered earlier.

2.8.1 Main Differenes Between Perceptron and Adaline

Observation 2.8.3 (**Perceptron vs Adeline**)
The diagram shows that the *Adaline algorithm compares the true class labels with the linear activation function's continuous-valued output to compute the model error and update the weights*, whereas the perceptron compares the true class labels to the predicted class labels.

2.8.2 Minimizing Loss Function with Gradient Descent

One of the key ingredients of supervised machine learning algorithms is a *defined objective function that is optimized during learning.* This *objective function is often a loss or cost function that we want to minimize.*

Observation 2.8.4 (**Loss Function in Adeline**)
In the case of Adaline, the **loss function**, L, can be *defined as the mean squared error (MSE) between the calculated outcome and the true class label*:

$$L(w, b) = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - \sigma \left(z^{(i)} \right) \right)^2$$

It turns out that the continuous linear activation function makes the loss function differentiable.

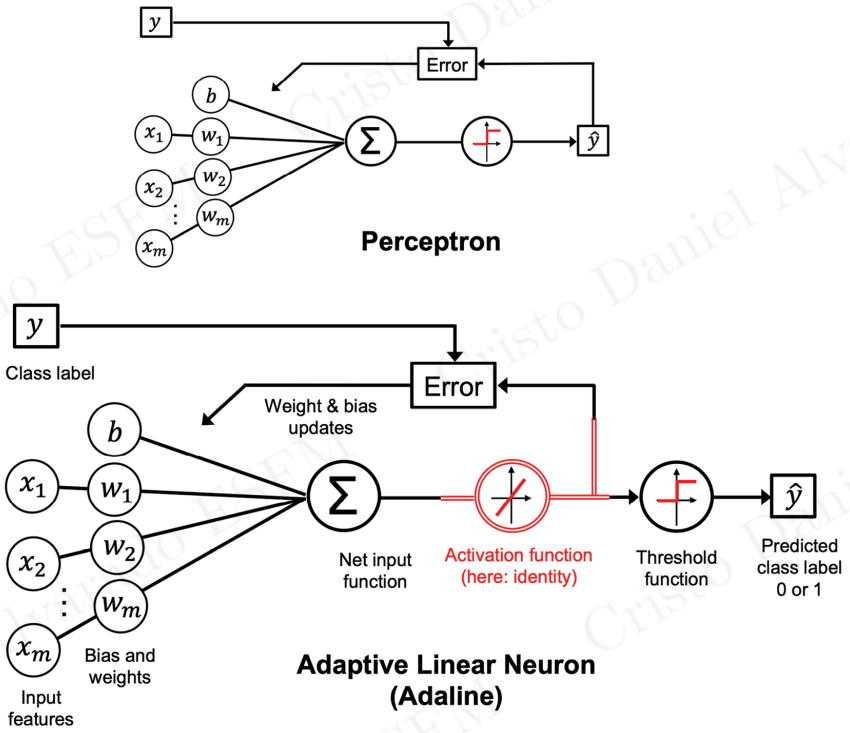

Figure 2.6: Perceptron vs Adeline Work Flux.

Also, this function is **convex**.

Definition 2.8.1 (**Convex Function**)
Let X be a convex subset of a real vector space, and $f : X \to \mathbb{R}$ a function. f is called **convex** if for all $x_1, x_2 \in X$ we have that:

$$f\left(tx_1 + (1-t)x_2\right) \leq tf\left(x_1\right) + (1-t)f\left(x_2\right), \quad \forall t \in [0,1]$$

The following image represents what a convex continuous function should look like:

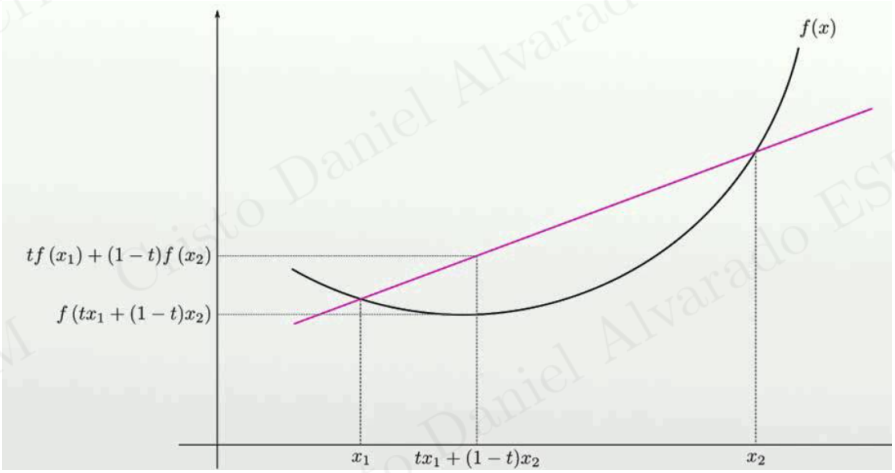

Figure 2.7: Convex Function.

Observation 2.8.5 (**About Convex Functions**)
When having a convex function and the inequality of the latter definition, the right side represents a straight line between $f(x_1)$ and $f(x_2)$. The argument of the function represents the line joining x_1 and x_2, so we compute its image under f.

Due to the fact that the loss function is convex, we have a simple yet powerful optimization algorithm called **gradient descent** can be *used to find the weights that minimize the loss function when classifying examples in the Iris dataset.*

Idea 2.8.2 (**Gradient Descent**)
The idea behind **gradient descent** is to *climb down a hill until a local or global loss minimum is reached.*

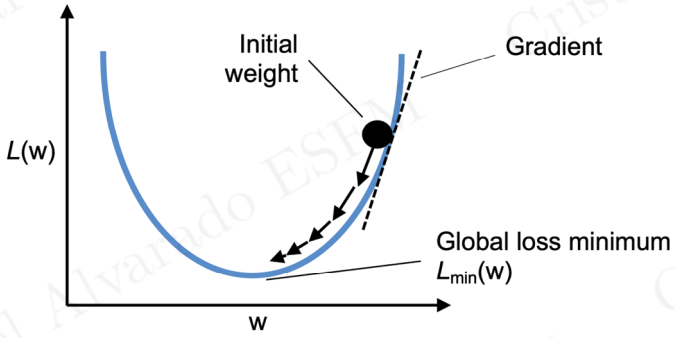

Figure 2.8: Gradient Descent.

The idea behind the algorithm is that on each iteration, a step is taken in the opposite direction of the gradient, where the step size is determined by the learning rate value and the slope of the gradient.

Observation 2.8.6 (**How Does Gradient Descent Works?**)
Gradient descent *updates each parameter by moving it in the direction that reduces the loss*:

$$\vartheta - \nabla_{\vartheta} L(\vartheta) \to \vartheta$$

Here, ϑ represents any parameter of the model. But since we have two different parameters w and b, we need to compute their individual gradients.

Using gradient descent, the model parameters are updated by taking a step in the opposite direction of the gradient, $\nabla L(w, b)$ of the loss function $L(w, b)$:

$$w = w + \Delta w$$
$$b = b + \Delta b$$

The parameter changes, Δw and Δb are defined as the negative gradient multiplied by the learning rate η:

$$\Delta w = -\eta \nabla_w L(w, b)$$
$$\Delta b = -\eta \nabla_b L(w, b)$$

To compute the gradient of the loss function, compute the partial derivative of the loss function with respect to each weight, w_j:

$$\frac{\partial L}{\partial w_j} = -\frac{2}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma(z^{(i)})\right)x_j^{(i)}$$

Similarly, compute the partial derivative of the loss with respect to the bias:

$$\frac{\partial L}{\partial b} = -\frac{2}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma(z^{(i)})\right)$$

So, the weight updates becomes:

$$\Delta w_j = -\eta\frac{\partial L}{\partial w_j}$$
$$\Delta b = -\eta\frac{\partial L}{\partial b}$$

Observation 2.8.7
Let's recall that $L : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$, so its gradient with respect to w and b will be:

$$\nabla_w L(w,b) = \begin{bmatrix} \frac{\partial L}{\partial w_1}(w,b) \\ \frac{\partial L}{\partial w_2}(w,b) \\ \vdots \\ \frac{\partial L}{\partial w_j}(w,b) \\ \vdots \\ \frac{\partial L}{\partial w_n}(w,b) \end{bmatrix}$$
$$\nabla_b L(w,b) = \left[\frac{\partial L}{\partial b}(w,b)\right] = \frac{\partial L}{\partial b}(w,b)$$

respectively. This makes perfect sense with the latter observation.

Note: The MSE derivative involves calculus for weight adjustments.

Observation 2.8.8 (**Batch Gradient Descent**)
Although the Adaline learning rule looks identical to the perceptron rule, $\sigma(z^{(i)})$ where $z^{(i)} = w^T x^{(i)} + b$ is a real number, not an integer class label.

Furthermore, the *weight update is calculated based on all examples in the training dataset (instead of updating the parameters incrementally after each training example), which is why this approach is often called* **batch gradient descent**. To be explicit and avoid confusion when discussing related concepts later in this section and throughout this course, this process is referred to as full **batch gradient descent**.

2.9 Implementing Adaline in Python

Since the perceptron rule and Adaline are very similar, we will take the perceptron implementation defined earlier and change the `fit` method so that the *weight and bias parameters are now updated by minimizing the loss function via gradient descent*:

```
class AdalineGD:
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    random_state : int
        Random number generator seed for random weight
          initialization.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    b_ : Scalar
        Bias unit after fitting.
    losses_ : list
      Mean squared error loss function values in each epoch.
    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
            Training vectors, where n_examples
            is the number of examples and
            n_features is the number of features.
        y : array-like, shape = [n_examples]
            Target values.

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=X.shape[1])
        self.b_ = np.float_(0.)
        self.losses_ = []

        for i in range(self.n_iter):
```

```
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.
               shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (errors**2).mean()
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def activation(self, X):
        """Compute linear activation"""
        return X

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(self.net_input(X))
                        >= 0.5, 1, 0)
```

Code 2.7: Adaline Implementation in Python.

Observation 2.9.1 (**Key Difference Between Adaline and Perceptron**)
Instead of updating the weights after evaluating each individual training example, as in the perceptron, **we calculate the gradient based on the whole training dataset**.

For the bias unit, *this is done via* **self.eta * 2.0 * errors.mean()**, where **errors is an array containing the partial derivative values** $\frac{\partial L}{\partial b}$ Similarly, we update the weights. However, note that the weight updates via the partial derivatives $\frac{\partial L}{\partial w_j}$ involve the feature values x_j, which we can compute by multiplying `errors` with each feature value for each weight:

```
for w_j in range(self.w_.shape[0]):
    self.w_[w_j] += self.eta * \
        (2.0 * (X[:, w_j]*errors)).mean()
```

Code 2.8: Computation of Partial Derivatives

To implement the weight update more efficiently without using a for loop, we can use a matrix-vector multiplication between the feature matrix and the error vector instead:

```
self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
```

Code 2.9: Weight Update Optimization.

Observation 2.9.2 (**Activation Method**)
The `activation` method *has no effect on the code since it is simply an identity function.* It *is included to illustrate how information flows through a single-layer neural network: features from the input data, net input, activation, and output.*

Later, we will learn about a logistic regression classifier that uses a *non-identity, nonlinear activation function.*

Observation 2.9.3 (**Logistic Regression Model**)
A **logistic regression model** *is closely related to Adaline, with the only difference being its activation and loss function.*

Similar to the previous perceptron implementation, we collect the loss values in a `self.losses_list` to *check whether the algorithm converges after training.*

Observation 2.9.4 (Note)
Matrix multiplication uses a vectorized approach for efficient computation.

In practice, it often requires some experimentation to find a good learning rate, η, for optimal convergence. Let's choose two different learning rates, $\eta = 0.1$ and $\eta = 0.0001$, and plot the loss values versus the number of epochs to see how well the Adaline implementation learns from the training data.

2.9.1 Hyperparameters

The learning rate, η (`eta`), and the number of epochs (`n_iter`) are hyperparameters of the perceptron and Adaline learning algorithms. Various techniques can help automatically find hyperparameter values that yield optimal classifier performance.

Let's now plot the loss against the number of epochs for the two different learning rates:

```
> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
...             np.log10(ada1.losses_), marker='o')
> ax[0].set_xlabel('Epochs'). ax[0].set_ylabel('log(Mean squared
    error)').
...             ada2.losses_, marker='o')
Set labels and title for the Adaline plot, then display it.
```

Code 2.10: Loss for Two Different Learning Rates.

Using a *learning rate that is too large can cause the mean squared error (MSE) to increase in every epoch because the updates overshoot the global minimum.* Conversely, *a learning rate that is too small can lead to very slow convergence, requiring a large number of epochs to reach the minimum loss.*

2.10 Improving Gradient Descent Through Feature Scaling

Many machine learning algorithms require some sort of feature scaling for optimal performance.

Definition 2.10.1 (**Feature Scaling**)
Feature Scaling is a crucial data preprocessing step in machine learning. Its primary goal is to normalize the range of independent variables or features of your data.

Example 2.10.1 (**Feature Scaling**)
When your dataset contains features that are on vastly different scales (e.g., age (0-100) and annual salary ($30,000 - $200,000)), many machine learning algorithms can behave poorly. Feature

scaling transforms these features onto a similar, comparable scale

Observation 2.10.1 (**Feature Scaling**)
Gradiant descent is one of the many algorithms that benefit from **feature scaling**.

Definition 2.10.2 (**Standarization**)
Standarization *is a normalization procedure that helps gradient descent learning converge more quickly*; however it *does not make the original dataset normaly distributed.*

Standarization shifts the mean of each feature so that it is centered at zero, and each feature has a standar deviation of 1 (unit variance).

Example 2.10.2 (**Standarization Procedure**)
For instance, to standarize the j-th feature, substract the sample mean, μ_j from every training example and divide it by its standar deviation, σ_j:

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Here, x_j is a vector consistent of the j-th feature values of all training examples, n, and and this standarization techinque applied to each feature, j in the dataset.

Observation 2.10.2 (Use of Standarization)
One reason why standardization helps with gradient descent learning is that it **becomes easier to find a learning rate that works well for all weights** (and the bias). *If the features are on vastly different scales, a learning rate that works well for updating one weight might be too large or too small to update another weight equally well. Using standardized features stabilizes the training so that the optimizer requires fewer steps to find a good or optimal solution* (**the global loss minimum**).

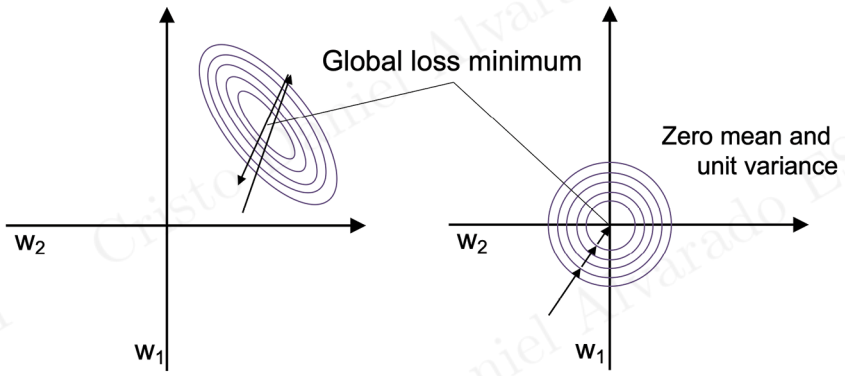

Figure 2.9: Global Loss Minimum with and without Zero Variance.

Standardization can easily be achieved by using the built-in NumPy methods `mean` and `std`:

```
X_std = np.copy(X)
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()code
```

Code 2.11: Standarization Using Numpy

After standardization, train Adaline again; it now converges after a small number of epochs using a learning rate of $\eta = 0.5$:

```
ada_gd = AdalineGD(n_iter=20, eta=0.5)
ada_gd.fit(X_std, y)
plot_decision_regions(X_std, y, classifier=ada_gd)
plt.title('Adaline - Gradient descent')
plt.xlabel('Sepal length [standardized]')
plt.ylabel('Petal length [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
plt.plot(range(1, len(ada_gd.losses_) + 1),
         ada_gd.losses_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Mean squared error')
plt.tight_layout()
plt.show()
```

Code 2.12: Adaline With Standarization Procedure

Executing this code displays the decision regions and the declining loss.

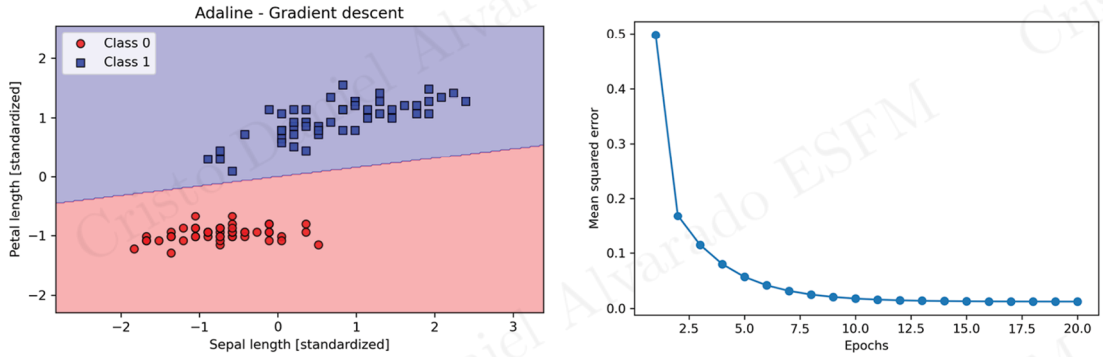

Figure 2.10: Adeline after Standarization Procedure

After training on the standardized features, Adaline has converged. However, the MSE remains non-zero even though all flower examples were classified correctly.

2.10.1 Large-Scale Machine Learning and Stochastic Gradient Descent

Previously, we minimized a loss function by taking a step in the opposite direction of the loss gradient that is calculated from the whole training dataset; this approach is sometimes referred to as **full batch gradient descent**.

Observation 2.10.3 (**Problem with Full Batch Gradient Descent**)
For datasets with millions of data points, *running full batch gradient descent can be computationally costly because the whole training dataset must be re-evaluated each time a step toward the global minimum is taken.*

A popular alternative to batch gradient descent is **stochastic gradient descent** (**SGD**), sometimes *called iterative or online gradient descent. Instead of updating the weights based on the sum of the accumulated errors over all training examples*, $x^{(i)}$:

$$\Delta w_j = \frac{2\eta}{n}\sum_{i=1}^{n}(y^{(i)} - \sigma(z^{(i)}))x_j^{(i)}$$

We update the parameters incrementally for each training example:

$$\Delta w_j = \eta(y^{(i)} - \sigma(z^{(i)}))x_j^{(i)}, \quad \text{and} \quad \Delta b = \eta(y^{(i)} - \sigma(z^{(i)}))$$

Observation 2.10.4
Although SGD can be considered an approximation of gradient descent, it typically reaches convergence much faster because of the more frequent weight updates.

Since each gradient is calculated from a single training example, the error surface is noisier than in gradient descent, which can help SGD escape shallow local minima when working with nonlinear loss functions. To obtain good results with SGD, present training data in a random order and shuffle the training dataset for every epoch to prevent cycles.

Observation 2.10.5 (**Note**)
Use an adaptive learning rate for better training results

Idea 2.10.1 (**Advantages of SGD**)
Another advantage of SGD is that *it supports online learning.* In online learning, *the model is trained on the fly as new training data arrives.* This is **especially useful when accumulating large amounts of data**, for example, customer data in web applications. The *system can immediately adapt to changes, and the training data can be discarded after updating the model if storage space is an issue.*

Observation 2.10.6 (**Note**)
Mini-batch gradient descent speeds up convergence with frequent updates.

Since the Adaline learning rule using gradient descent is already implemented, only minor adjustments are needed to modify the learning algorithm for weight updates via SGD.

Inside the `fit` method, weights are updated after each training example. An additional `partial_fit` method, which does not reinitialize the weights, supports online learning. To check whether the algorithm converged after training, the average loss of the training examples is calculated for each epoch. An option to shuffle the training data before each epoch avoids repetitive cycles when optimizing the loss function, and the `random_state` parameter allows specification of a random seed for reproducibility:

```
"""ADAptive LInear NEuron classifier.

Parameters
------------
eta : float
    Learning rate (between 0.0 and 1.0)
n_iter : int
    Passes over the training dataset.
shuffle : bool (default: True)
    Shuffles training data every epoch if True to prevent
    cycles.
random_state : int
    Random number generator seed for random weight
    initialization.

Attributes
-----------
w_ : 1d-array
    Weights after fitting.
b_ : Scalar
    Bias unit after fitting.
losses_ : list
    Mean squared error loss function value averaged over all
    training examples in each epoch.

"""
def __init__(self, eta=0.01, n_iter=10,
             shuffle=True, random_state=None):
    self.eta = eta
    self.n_iter = n_iter
    self.w_initialized = False
    self.shuffle = shuffle
    self.random_state = random_state

def fit(self, X, y):
    """ Fit training data.

    Parameters
    ----------
    X : {array-like}, shape = [n_examples, n_features]
        Training vectors, where n_examples is the number of
        examples and n_features is the number of features.
    y : array-like, shape = [n_examples]
        Target values.

    Returns
    -------
    self : object

    """
    self._initialize_weights(X.shape[1])
```

```
52      self.losses_ = []
53      for i in range(self.n_iter):
54          if self.shuffle:
55              X, y = self._shuffle(X, y)
56          losses = []
57          for xi, target in zip(X, y):
58              losses.append(self._update_weights(xi, target))
59          avg_loss = np.mean(losses)
60          self.losses_.append(avg_loss)
61      return self
62
63  def partial_fit(self, X, y):
64      """Fit training data without reinitializing the weights"""
65      if not self.w_initialized:
66          self._initialize_weights(X.shape[1])
67      if y.ravel().shape[0] > 1:
68          for xi, target in zip(X, y):
69              self._update_weights(xi, target)
70      else:
71          self._update_weights(X, y)
72      return self
73
74  def _shuffle(self, X, y):
75      """Shuffle training data"""
76      r = self.rgen.permutation(len(y))
77      return X[r], y[r]
78
79  def _initialize_weights(self, m):
80      """Initialize weights to small random numbers"""
81      self.rgen = np.random.RandomState(self.random_state)
82      self.w_ = self.rgen.normal(loc=0.0, scale=0.01,
83                                 size=m)
84      self.b_ = np.float_(0.)
85      self.w_initialized = True
86
87  def _update_weights(self, xi, target):
88      """Apply Adaline learning rule to update the weights"""
89      output = self.activation(self.net_input(xi))
90      error = (target - output)
91      self.w_ += self.eta * 2.0 * xi * (error)
92      self.b_ += self.eta * 2.0 * error
93      loss = error**2
94      return loss
95
96  def net_input(self, X):
97      """Calculate net input"""
98      return np.dot(X, self.w_) + self.b_
99
100 def activation(self, X):
101     """Compute linear activation"""
102     return X
103
```

```
104 def predict(self, X):
105     """Return class label after unit step"""
106     return np.where(self.activation(self.net_input(X))
107                     >= 0.5, 1, 0)
```

Code 2.13: Adaline Implemented Using Stochastic Gradient Descent

The `_shuffle` method uses `np.random.permutation` to generate a random sequence of unique indices, which are then applied to shuffle the feature matrix and class-label vector.

Observation 2.10.7
Display the plot with labeled axes using `plt.show()`.

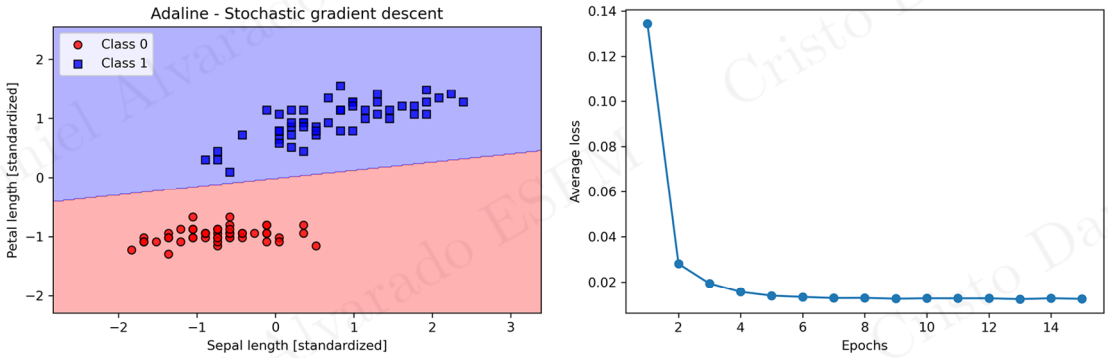

Figure 2.11: Adaline Implementation using Stochastic Gradient DEscent

The *average loss decreases quickly, and the final decision boundary after 15 epochs is similar to the batch gradient descent Adaline.*

Idea 2.10.2 (Updating the Model in an Online Scenario)
To update the model in an online learning scenario with streaming data, call `ada_sgd.partial_fit(X_std[0, :], y[0])` on individual training examples.

Chapter 3

A Tour of Machine Learning Classifiers Using Scikit-Learn

3.1 Introduction

The goal of this chapter is to use Scikit-Learn to impelement some classifiers in order to learn how to use them, his advantages, disadvantages and different use scenarios.

Specifically, we will take a look of 4 popular machine learning models commonly used in academia and in industry. In addition, we will take a look at the Scikit-Learn library, which offers a user-friendly and consistent interface for using those algorithms efficiently and productively.

3.1.1 Choosing a Classification Algorithm

Each algorithm has its own quirks and relies on certain assumptions. no single classifier works best across all possible scenarios (The Lack of A Priori Distinctions Between Learning Algorithms, Wolpert, David H, Neural Computation 8.7 (1996): 1341-1390).

Observation 3.1.1 (**Comparasion**)
In practice, it is always recomended to compare the behaviour of different algorithms in order to find the best model suitable for a particular problem; these may differ in the number of features or examples, the amount of noise in a dataset, and whether the classes are linearly separable.

Idea 3.1.1
The performance of a classifier relies upon the data that is available for learning. The five main steps that are involved in training a supervised machine learning algorithm can be summarized as follows:

1. Selecting features and collecting labeled training examples
2. Choosing a performance metric
3. Choosing a learning algorithm and training a model
4. Evaluating the performance of the model
5. Changing the settings of the algorithm and tuning the model.

We will mainly focus on the main concepts of the different algorithms in this chapter and revisit topics such as feature selection and preprocessing, performance metrics, and hyperparameter tuning for more detailed discussions later in the book.

3.1.2 First Steps with scikit-learn Training a Perceptron

Before we learn about two related learning algorithms: the perceptron and adaline, both implemented in Python using NumPy and other libraries by ourselves.

Now we will take a look at the **scikit-learn API**.

Observation 3.1.2
One of the advantages of using scikit-learn is that combines a user-friendly and consistent interface with a highly optimized implementation of several classification algorithms. Also, this library offers a not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models.

3.1.3 Training a Model Using Scikit-Learn

To get started with the scikit-learn library, we will train a perceptron model similar to the one that we implemented in Chapter 2. For simplicity, we will use the already familiar Iris dataset throughout the following sections.

Observation 3.1.3 (**Iris Dataset and Its Uses**)
Conveniently, the Iris dataset is already available via scikit-learn, since it is a simple yet popular dataset that is frequently used for testing and experimenting with algorithms. Similar to the previous chapter, we will only use two features from the Iris dataset for visualization purposes.

We will assign the petal length and petal width of the 150 flower examples to the feature matrix, `X`, and the corresponding class labels of the flower species to the vector array, `y`:

```
1 Class labels: [0 1 2]
2 Class labels: [0 1 2]
```

Code 3.1: Class Labels of Matrix `X`

The `np.unique(y)` function returned the three unique class labels stored in `iris.target`, and as we can see, the Iris flower class names, `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`, are already stored as integers (here: `0`, `1`, `2`).

Observation 3.1.4 (**Scikit-Learn with Class Labels and String Formats**)
Although many scikit-learn functions and class methods also work with *class labels in string format, using integer labels is a recommended approach to avoid technical glitches and improve computational performance due to a smaller memory footprint*; furthermore, *encoding class labels as integers is a common convention among most machine learning libraries.*

To evaluate how well a trained model performs on unseen data, we will *further split the dataset into separate training and test datasets.*

Idea 3.1.2 (**More Info About Best Practices around Model Evaluation**)
In Chapter 6, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we **will discuss the best practices around model evaluation in more detail**.

Using the `train_test_split` function from scikit-learn's `model_selection` module, we randomly split the `X` and `y` arrays into 30 percent test data (45 examples) and 70 percent training data (105 examples):

```
train_test_split(X, y, test_size=0.3, random_state=1, stratify=y
    )
```

Code 3.2: Train Test Split Function

Observation 3.1.5
Note that the `train_test_split` function already shuffles the training datasets internally before splitting; otherwise, all examples from class `0` and class `1` would have ended up in the training datasets, and the test dataset would consist of 45 examples from class `2`. Via the `random_state` parameter, we provided a fixed random seed (`random_state=1`) for the internal pseudo-random number generator that is used for shuffling the datasets prior to splitting. Using such a fixed `random_state` ensures that our results are reproducible.

Lastly, we took advantage of the built-in support for stratification via `stratify=y`. In this context, *stratification means that the* `train_test_split` *method returns training and test subsets that have the same proportions of class labels as the input dataset.* We can use NumPy's `bincount` function, which counts the number of occurrences of each value in an array, to verify that this is indeed the case:

```
>>> print('Labels counts in y:', np.bincount(y))
Labels counts in y: [50 50 50]
>>> print('Labels counts in y_train:', np.bincount(y_train))
Labels counts in y_train: [35 35 35]
>>> print('Labels counts in y_test:', np.bincount(y_test))
Labels counts in y_test: [15 15 15]
```

Code 3.3: Use of `bincount` Method in Python.

Idea 3.1.3 (Feature Scaling)
Many machine learning and optimization algorithms also require **feature scaling for optimal performance**, as we saw in the gradient descent example in Chapter 2. Here, **we will standardize the features using the StandardScaler class from scikit-learn's preprocessing module**:

```
sc = StandardScaler()
```

Code 3.4: `StandardScaler` Method in Python.

Using the preceding code, we loaded the `StandardScaler` class from the `preprocessing` module and initialized a new `StandardScaler` object that we assigned to the `sc` variable.

Using the `fit` method, `StandardScaler` estimated the parameters, μ (sample mean) and σ (standard deviation), for each feature dimension from the training data. By calling the `transform` method, we then standardized the training data using those estimated parameters, μ and σ. Note that we used the same scaling parameters to standardize the test dataset so that both the values in the training and test dataset are comparable with one another.

Example 3.1.1
Having standardized the training data, we can now train a perceptron model. Most algorithms in scikit-learn already support multiclass classification by default via the one-versus-rest (OvR) method, which allows us to feed the three flower classes to the perceptron all at once.

Finally, we have the code for the model:

```
from sklearn import datasets #To import Iris Dataset
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import numpy as np
import sklearn.linear_model as linear_models
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

#Plot Regions

def plot_decision_regions(X, y, classifier, test_idx=None,
    resolution=0.2):
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution)
        ,
                           np.arange(x2_min, x2_max, resolution)
                            )
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()
        ]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')

    if test_idx is not None:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='Test set')

```

```
#Import Data

iris = datasets.load_iris()
X = iris.data[:,2:4]
y = iris.target

#Scale Standard Data

sc = StandardScaler()
X = sc.fit_transform(X)

#Import Perceptron

clf = linear_models.Perceptron(eta0=0.1, random_state=1)

# Split training data and test data
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

#Fit model with training data

clf.fit(X_train, y_train)

#Plot regions

plot_decision_regions(X, y, clf, idx_test, 0.2)
plt.xlabel(iris.feature_names[2] + " (standarized)")
plt.ylabel(iris.feature_names[3] + " (standarized)")
plt.legend()
plt.show()
```

Code 3.5: Perceptron Trained Model using Sklearn

With the slight modification that we made to the `plot_decision_regions` function, we can now specify the indices of the examples that we want to mark on the resulting plots.

As we can see in the resulting plot, the three flower classes can't be perfectly separated by a linear decision boundary:

However, remember from our discussion in Chapter 2 that **the perceptron algorithm never converges on datasets that aren't perfectly linearly separable**, which is *why the use of the perceptron algorithm is typically not recommended in practice*. In the following sections, we will look at more powerful linear classifiers that converge to a loss minimum even if the classes are not perfectly linearly separable.

Observation 3.1.6 (**Note**)
The Perceptron, as well as other scikit-learn functions and classes, *often has additional parameters that we omit for clarity. You can read more about those parameters using the help function in Python (for instance, `help(Perceptron)`) or by going through the excellent scikit-learn online*

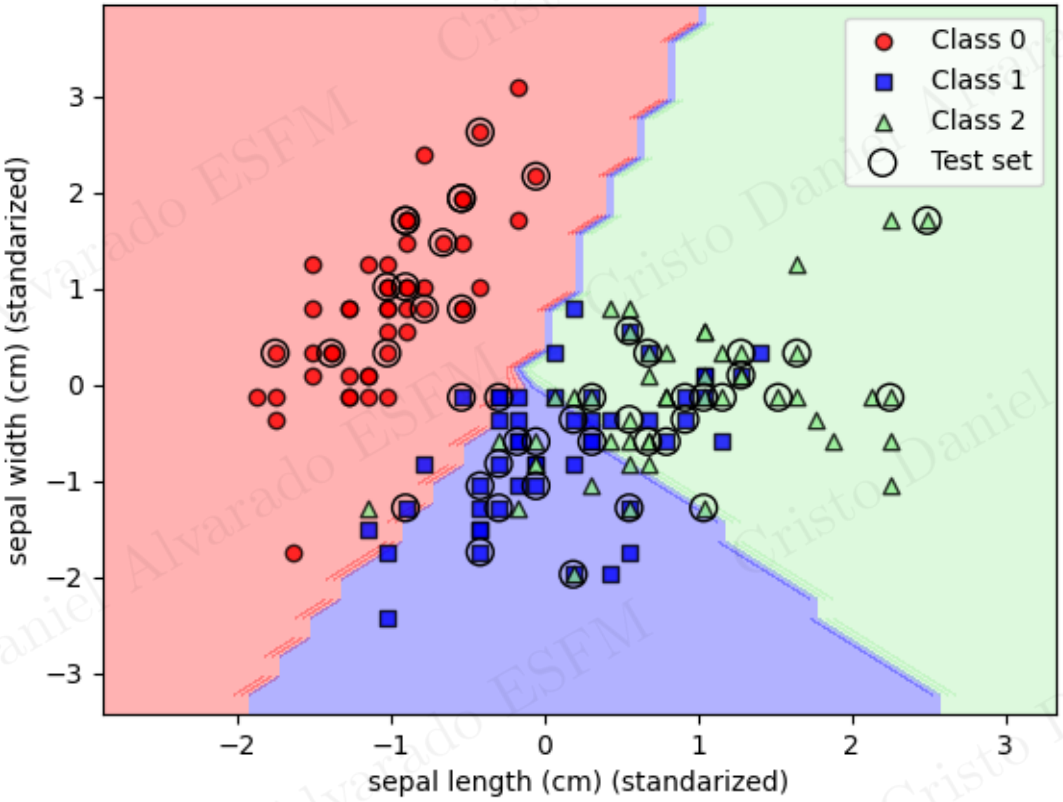

Figure 3.1: Perceptron Trained with Iris Dataset using Scikit-Learn.

documentation here.

3.2 Modeling Class Probabilities Via Logistic Regression

The biggest disadvantage of the Perceptron is that it never converges if the classes aren't linearly separable. The reason for this is that the weights are continuously updated because there is always at least one misclassified training example present in each poch.

To make better use of our time, we will now look at another simple, yet more powerful, algorithm for linear and binary classification problems: **logistic regression**. Note that, despite its name, *logistic regression is a model for classification, not regression.*

3.2.1 Logistic Regression and Conditional Probabilities

Definition 3.2.1 (**Logistic Regression**)
Logistic regression is a *classification model that is easy to implement and performs very well on linearly separable classes.* It is *one of the most widely used algorithms for classification in industry.*

Similar to the perceptron and Adaline, *the logistic regression model in this section is also a linear model for binary classification.*

Idea 3.2.1 (**Note**)
Logistic regression can be generalized to multinomial logistic regression.

To explain the main mechanics behind logistic regression as a probabilistic model for binary classification, let's first introduce the odds: the odds in favor of a particular event. The odds can be written as $\frac{p}{1-p}$, where p stands for the probability of the positive event. The term "positive event" does not necessarily mean "good," but refers to the event that we want to predict.

Example 3.2.1 (**Conditional Probability**)
Consider the **probability that a patient has a certain disease given certain symptoms**; we can think of the positive event as class label $y = 1$ and the symptoms as features $\boldsymbol{x}$. Hence, for brevity, we can define the probability p as $p := p(y = 1|\boldsymbol{x})$, the conditional probability that a particular example belongs to class 1 given its features, $\boldsymbol{x}$.

We can then further define the logit function, which is simply the logarithm of the odds (log-odds):

$$\text{logit}\,(p) = \log\left(\frac{p}{1-p}\right) \tag{3.1}$$

Note that log *refers to the natural logarithm, as it is the common convention in computer science.* The logit function *takes input values in the range 0 to 1 and transforms them into values over the entire real-number range.*

Under the logistic model, we **assume that there is a linear relationship between the weighted inputs and the log-odds**:

$$\text{logit}\,(p) = w_1x_1 + \cdots + w_nx_n + b = \sum_{i=1}^{n} w_ix_i + b = \boldsymbol{w}^T\boldsymbol{x} + b \tag{3.2}$$

Observation 3.2.1
While the preceding describes an assumption we make about the linear relationship between the log-odds and the net inputs, what **we are actually interested in is the probability** p, the *class-membership probability of an example given its features.* While the logit function maps the probability to a real-number range, *we can consider the inverse of this function to map the real-number range back to a* $[0, 1)$ range for the probability p.

This inverse of the logit function is typically called the **logistic sigmoid function**, which is *sometimes simply abbreviated to sigmoid function due to its characteristic S-shape*:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \forall z \in \mathbb{R} \tag{3.3}$$

Here, z is the *net input, the linear combination of weights and the inputs* (that is, the features associated with the training examples):

$$z = \boldsymbol{w}^T \boldsymbol{x} + b \tag{3.4}$$

Example 3.2.2 (**Graph of Sigmoid Function**)
Graph of Sigmoid Function:

```
import matplotlib.pyplot as plt
import numpy as np

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

z = np.arange(-7, 7, 0.1)
sigma_z = sigmoid(z)
plt.plot(z, sigma_z)
plt.axvline(0.0, color='k')
plt.ylim(-0.1, 1.1)
plt.xlabel('z')
plt.ylabel('$\\sigma (z)$')
# y-axis ticks and gridline
plt.yticks([0.0, 0.5, 1.0])
ax = plt.gca()
ax.yaxis.grid(True)
plt.tight_layout()
plt.show()
```

Code 3.6: Sigmoid Function Graph.

As a result of executing the previous code, you should now see the S-shaped (sigmoidal) curve.

Observation 3.2.2 (**Comparasion with Adaline**)
To build some understanding of the logistic regression model, we can relate it to Adaline. In Adaline, we used the identity function, $\sigma(z) = z$ as the activation function. In logistic regression, *this activation function simply becomes the sigmoid function defined earlier.*

The only difference between Adaline and logistic regression is the activation function.

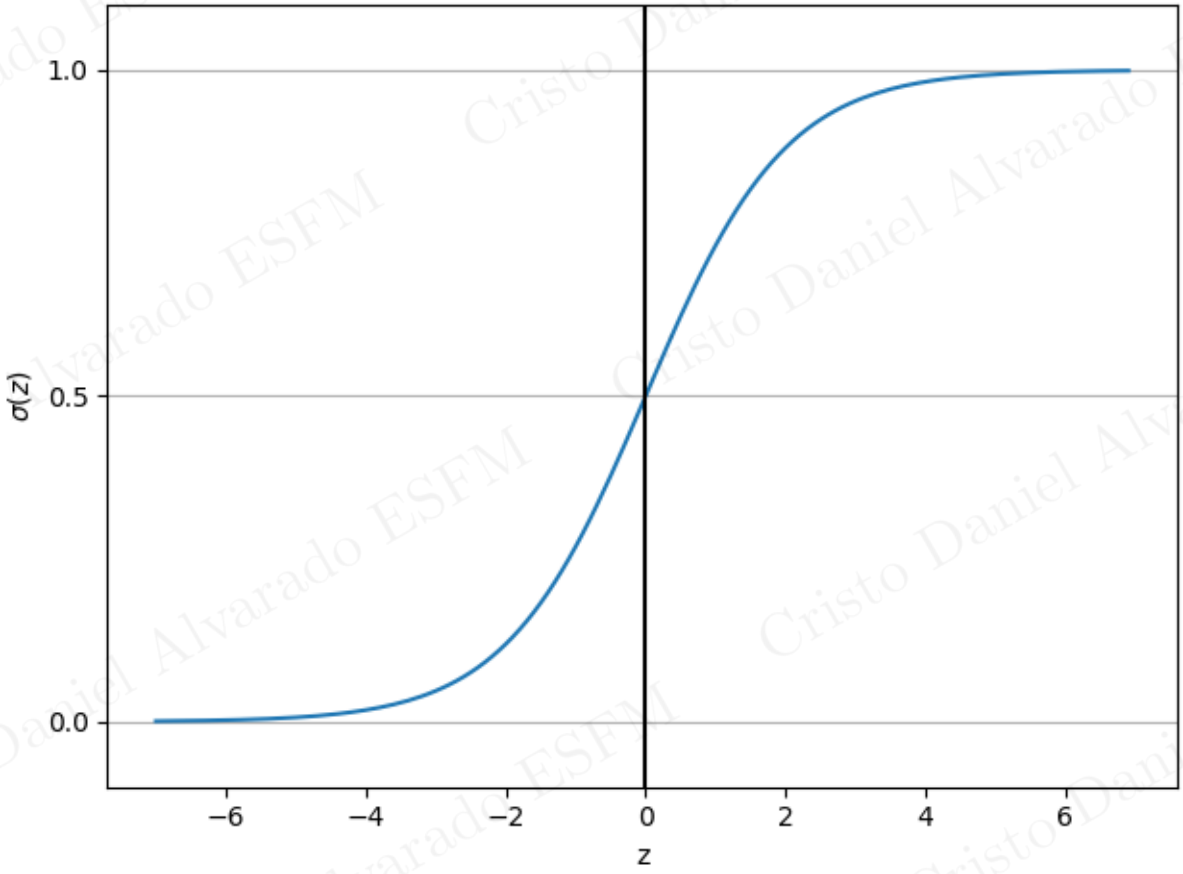

Figure 3.2: Sigmoid Function Graph.

Definition 3.2.2 (**Purpose of an Activation Function**)
In simple terms, the purpose of an **activation function** is *to decide whether a neuron in a neural network should be "activated" or not, and to what degree. It's what allows neural networks to learn and model complex, non-linear relationships.*

The output of the sigmoid function is then interpreted as the probability of a particular example belonging to class 1, $\sigma(z) = p(y = 1|\boldsymbol{x}; \boldsymbol{w}, b)$, given its features $\boldsymbol{x}$, and parameterized by the weights and bias, $\boldsymbol{w}$ and b.

For example, if we compute $\sigma(z) = 0.8$ for a particular flower, it means that the chance that this example is an `Iris-versicolor` flower is 80 percent. Therefore, the probability that this flower is an `Iris-setosa` can be calculated as $p(y = 0|\boldsymbol{x}; \boldsymbol{w}, b) = 1 - p(y = 1|\boldsymbol{x}; \boldsymbol{w}, b) = 0.2$, or 20 percent.

The predicted probability can then be converted into a binary outcome via a threshold function:

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

If we look at the preceding plot of the sigmoid function, this is equivalent to the following:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Observation 3.2.3 (**Interest in Predicted Class Labels**)
In many applications, **we are not only interested in the predicted class labels but also in the class-membership probabilities produced by the sigmoid function before applying the threshold**.

Logistic regression is *used in weather forecasting, for example, not only to predict whether it will rain on a particular day but also to report the chance of rain.* Similarly, logistic regression *can be used to predict the chance that a patient has a particular disease given certain symptoms, which is why logistic regression enjoys great popularity in the field of medicine.*

3.3 Learning the Model Weights via the Logistic Loss Function

You have learned how we can use the logistic regression model to predict probabilities and class labels; now, let's briefly talk about how *we fit the parameters of the model, for instance, the weights and bias unit,* $\boldsymbol{w}$ *and* b. Previously, we defined the mean squared error loss function as follows:

$$L(\boldsymbol{w}, b|\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma(z^{(i)})\right)^2$$

We minimized this function in order to learn the parameters for our Adaline classification model. To explain how we can derive the loss function for logistic regression, *let's first define the likelihood*, $\mathcal{L}$, that we **want to maximize when we build a logistic regression model**, assuming that the individual examples in our dataset are independent of one another. The formula is as follows:

$$L(w, b \mid x) = p(y|x; w, b) = \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; w, b) = \prod_{i=1}^{n} \left(\sigma(z^{(i)})\right)^{y^{(i)}} \left(1 - \sigma(z^{(i)})\right)^{(1-y^{(i)})} \tag{3.5}$$

This is the **likelihood function** $L(w, b \mid x)$ in *terms of a product over* n *terms.* It is expressed as the product of probabilities $p(y^{(i)} \mid x^{(i)}; w, b)$ for i from 1 to n, which is further expanded to the product of $(\sigma(z^{(i)}))^{y^{(i)}}$ times $(1 - \sigma(z^{(i)}))^{(1-y^{(i)})}$, where σ is the sigmoid function.

Observation 3.3.1
In practice, it is easier to maximize the (natural) log of this equation, which is called the log-likelihood function:

$$l(w, b \mid x) = \log\left(\mathcal{L}\left(w, b \mid x\right)\right) = \sum_{i=1}^{n}\left[y^{(i)}\log(\sigma(z^{(i)})) + (1 - y^{(i)})\log(1 - \sigma(z^{(i)}))\right] \tag{3.6}$$

Applying the log function reduces the potential for numerical underflow if the likelihoods are very small. In addition, the product of factors becomes a summation of factors, which makes it easier to obtain the derivative of this function via the addition trick.

3.3.1 Deriving the likelihood function

We can obtain the expression for the likelihood of the model given the data, $\mathcal{L}(w, b \mid x)$, as follows. Given that we have a binary classification problem with class labels 0 and 1, we can think of the label 1 as a Bernoulli variable—it can take on two values, 0 and 1, with the probability p of being 1: $Y \sim \text{Bernoulli}(p)$.

For a single data point, we can write this probability as:

$$P(Y = 1 \mid X = x^{(i)}) = \sigma(z^{(i)}) \tag{3.7}$$

$$P(Y = 0 \mid X = x^{(i)}) = 1 - \sigma(z^{(i)}) \tag{3.8}$$

Putting these two expressions together, and using the shorthand $P(Y = y^{(i)} \mid X = x^{(i)}) = p(y^{(i)} \mid x^{(i)})$, we get the probability mass function of the Bernoulli variable:

$$p\left(y^{(i)} \middle| x^{(i)}\right) = \left(\sigma(z^{(i)})\right)^{(y^{(i)})} \left(1 - \sigma(z^{(i)})\right)^{1-y^{(i)}}$$

Substituting the probability mass function of the Bernoulli variable, we arrive at the expression for the likelihood, which we attempt to maximize by changing the model parameters:

$$\mathcal{L}(w, b \middle| x) = \prod_{i=1}^{n} \left(\sigma(z^{(i)})\right)^{(y^{(i)})} \left(1 - \sigma(z^{(i)})\right)^{1-y^{(i)}}$$

Observation 3.3.2
We could use an optimization algorithm such as **gradient ascent** to maximize this log-likelihood function. (*Gradient ascent works the same way as gradient descent explained earlier, except that gradient ascent maximizes a function instead of minimizing it.*).

Alternatively, let's rewrite the log-likelihood as a loss function, L, that can be minimized using gradient descent:

$$L(w, b) = \sum_{i=1}^{n} \left[-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))\right]$$

Example 3.3.1
For one single training example, looking at the equation, we can see that the first term becomes zero if $y = 0$, and the second term becomes zero if $y = 1$:

$$L(\sigma(z), y, w, b) = \begin{cases} -\log\left(\sigma(z)\right) & \text{if } y = 0 \\ -\log\left(1 - \sigma(z)\right) & \text{if } y = 1 \end{cases}$$

Let's write a short code snippet to create a plot that illustrates the loss of classifying a single training example for different values of $\sigma(z)$. The resulting plot shows the sigmoid activation on the x axis in the range 0 to 1 (the inputs to the sigmoid function were z values in the range -10 to 10) and the associated logistic loss on the y axis:

We can see that the loss approaches 0 (continuous line) if we correctly predict that an example belongs to class 1. Similarly, the loss also approaches 0 if we correctly predict y = 0 (dashed line). However, if the prediction is wrong, the loss goes toward infinity. The main point is that we penalize incorrect predictions with an increasingly larger loss.

3.4 Converting an Adaline Implementation Into an Algorithm for Logistic Regression

If we implement logistic regression from scratch, we can substitute the loss function, L, in the earlier Adaline implementation with the new loss function:

$$L(w, b) = \frac{1}{n} \sum_{i=1}^{n} \left[-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))\right]$$

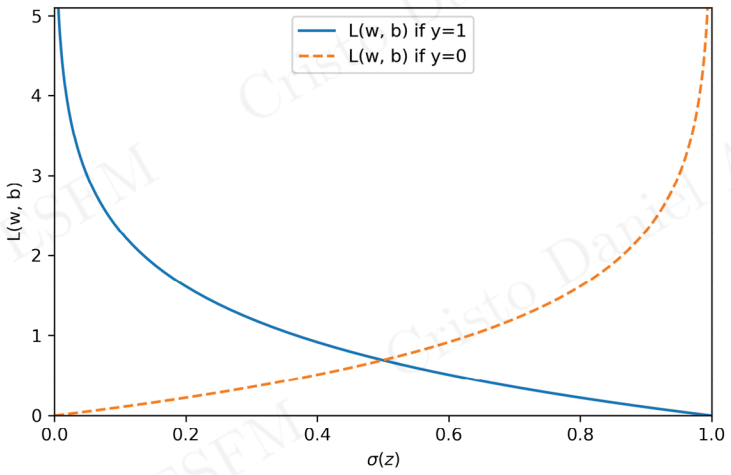

Figure 3.3: Loss Function.

Observation 3.4.1
We use this to compute the loss of classifying all training examples per epoch. We also swap the linear activation function for the sigmoid. With those changes, the Adaline code becomes a working logistic regression implementation.

The following example uses full-batch gradient descent (the same changes can be applied to the stochastic variant):

```
import numpy as np

class LogisticRegressionGD:
    """Gradient descent-based logistic regression classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    random_state : int
      Random number generator seed for random weight
      initialization.

    Attributes
    -----------
    w_ : 1d-array
      Weights after training.
    b_ : Scalar
      Bias unit after fitting.
    losses_ : list
      Mean squared error loss function values in each epoch.
    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
```

```
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
          Training vectors, where n_examples is the
          number of examples and n_features is the
          number of features.
        y : array-like, shape = [n_examples]
          Target values.

        Returns
        -------
        self : Instance of LogisticRegressionGD
        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape
           [1])
        self.b_ = np.float_(0.)
        self.losses_ = []
        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.
               shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (-y.dot(np.log(output))
- ((1 - y).dot(np.log(1 - output)))
                    / X.shape[0])
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def activation(self, z):
        """Compute logistic sigmoid activation"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(self.net_input(X)) >=
           0.5, 1, 0)
```

Code 3.7: Implementation of Logistic Regression in Python.

When fitting a logistic regression model, remember that it only works for binary classification tasks.

The resulting decision region plot looks as follows:

3.5 Gradient Descent

One observation we can make on the latter code is that the partial derivative of the loss function L is really simple to compute, this is due to the fact:

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_j}$$

where:

$$\frac{\partial L}{\partial \sigma} = \frac{\sigma - y}{\sigma \cdot (1 - \sigma)}$$
$$\frac{\partial \sigma}{\partial z} = \sigma(1 - \sigma)$$
$$\frac{\partial z}{\partial w_j} = x_j$$

So,

$$\frac{\partial L}{\partial w_j} = -(y - \sigma)x_j$$

Recall that gradient descent takes steps in the opposite direction of the gradient. Hence, we flip $\nabla L(w)$ and update the j-th weight as follows, including the learning rate η:

$$w_j = w_j - \eta \cdot \frac{\partial L}{\partial w_j}(w)$$

While the partial derivative of the loss function with respect to the bias unit is not shown, bias derivation follows the same concept.

Observation 3.5.1
We should compute the mean in all this process, but due we are working in this example with a single record for simplicity.

Idea 3.5.1 (**Relation between Adaline and Logistical Regression**)
Both the weight and bias updates are the same as for Adaline, but the change is in the activation function.

3.6 Training a Logistic Regression Model with Scikit-Learn

We just went through useful coding and math exercises in the previous subsection, which helped to illustrate the conceptual differences between Adaline and logistic regression. Now, let's learn how to use scikit-learn's more optimized implementation of logistic regression, which also supports multiclass settings off the shelf.

Observation 3.6.1
Note that in recent versions of scikit-learn, the technique used for multiclass classification—multinomial or OvR—is chosen automatically.

In the following code example, we will use the `sklearn.linear_model.LogisticRegression` class as well as the familiar fit method to train the model on all three classes in the standardized flower training dataset. Also, we set `multi_class='ovr'` for illustration purposes. As an exercise, you may want to compare the results with `multi_class='multinomial'`. Note that the `multinomial` setting is now the default choice in scikit-learn's `LogisticRegression` class and recommended in practice for mutually exclusive classes, such as those found in the Iris dataset.

Observation 3.6.2
Here, "mutually exclusive" means that each training example can belong to only a single class (in contrast to multilabel classification, where a training example can be a member of multiple classes).

Now, let's look at the code example:

```
from sklearn.linear_model import LogisticRegression
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets #To import Iris Dataset
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

def plot_decision_regions(X, y, classifier, test_idx=None,
    resolution=0.2):
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution)
        ,
                           np.arange(x2_min, x2_max, resolution)
                               )
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()
        ]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
```

```
                    label=f'Class {cl}',
                    edgecolor='black')

    if test_idx is not None:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='Test set')

#Import Data

iris = datasets.load_iris()
X = iris.data[:,2:4]
y = iris.target

#Scale Standard Data

sc = StandardScaler()
X = sc.fit_transform(X)

# Split training data and test data
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

lr = LogisticRegression(C=100.0, solver='lbfgs',
                        multi_class='ovr')
lr.fit(X_train, y_train)
plot_decision_regions(X, y, lr, idx_test, 0.01)

plt.xlabel(iris.feature_names[2] + " (standarized)")
plt.ylabel(iris.feature_names[3] + " (standarized)")
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

Code 3.8: Logistic Regression Iris Dataset Python.

Which produces an output like this:

3.6.1 Algorithms for convex optimization

Many different algorithms exist for solving optimization problems. For minimizing convex loss functions, such as the logistic regression loss, it is recommended to use more advanced approaches than regular **stochastic gradient descent** (**SGD**). Scikit-learn implements a range of such optimization algorithms, which can be specified via the solver parameter: `'newton-cg'`, `'lbfgs'`, `'liblinear'`, `'sag'`, and `'saga'`.

While the logistic regression loss is convex, most optimization algorithms should converge to the global loss minimum with ease. However, there are certain advantages to using one algorithm over

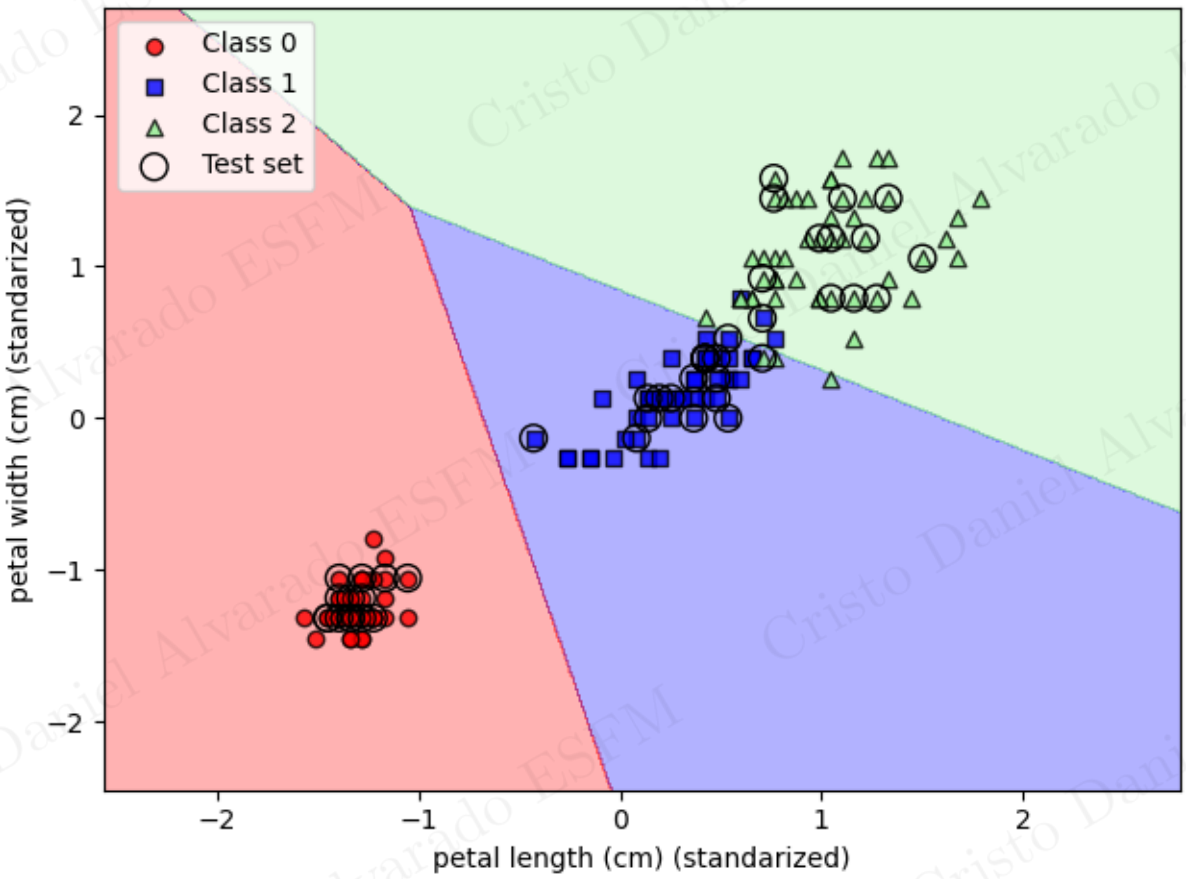

Figure 3.4: Logistic Regression on the Iris Dataset.

another. For example, in earlier versions (for instance, v0.21), scikit-learn used `'liblinear'` as the default, which cannot handle the multinomial loss and is limited to the OvR scheme for multiclass classification. In scikit-learn v0.22, the default solver was changed to `'lbfgs'`, which stands for the limited-memory Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm and is more flexible in this regard.

Observation 3.6.3
Looking at the preceding code that we used to train the `LogisticRegression` model, you might now be wondering, "What is this mysterious parameter `C`?" We will discuss this parameter in the next subsection, where we will introduce the concepts of overfitting and regularization. However, before we move on to those topics, let's finish our discussion of class membership probabilities.

The probability that training examples belong to a certain class can be computed using the `predict_proba` method.

Example 3.6.1
For example, we can predict the probabilities of the first three examples in the test dataset as follows:

```
1 lr.predict_proba(X_test[:3, :])
```

Code 3.9: Predicting Probabilities.

This code snippet returns the following array:

```
array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])
```

The first row corresponds to the class membership probabilities of the first flower, the second row corresponds to the second flower, and so forth. Notice that the column-wise sum in each row is 1, as expected (you can confirm this by executing `lr.predict_proba(X_test_std[:3, :]).sum(axis=1)`).

The highest value in the first row is approximately `0.85`, which means that the first example belongs to class 3 (`Iris-virginica`) with a predicted probability of 85 percent.

Observation 3.6.4

As you may have noticed, we can obtain the predicted class labels by identifying the largest column in each row, for example, by using NumPy's `argmax` function:

```
lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

The returned class indices (corresponding to `Iris-virginica`, `Iris-setosa`, and `Iris-setosa`) are:

```
array([2, 0, 0])
```

In the preceding code example, we converted conditional probabilities into class labels manually by using NumPy's `argmax` function. In practice, the more convenient way of obtaining class labels when using scikit-learn is to call the `predict` method directly:

```
lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

Lastly, if you want to predict the class label of a single flower example, scikit-learn expects a two-dimensional array as input. One way to convert a single row entry into a two-dimensional data array is to use NumPy's `reshape` method to add a new dimension, as shown here:

```
lr.predict(X_test[0, :].reshape(1, -1))
array([2])
```

3.7 Tackling Overfitting via Regularization

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from overfitting, we also say that the model has high variance, which can be caused by having too many parameters, leading to a model that is too complex given the underlying data. Similarly, a model can suffer from underfitting (high bias), which means it is not complex enough to capture the pattern in the training data well and therefore also performs poorly on unseen data.

Although we have only encountered linear models for classification so far, the problems of overfitting and underfitting can be illustrated by comparing a linear decision boundary to more complex, nonlinear decision boundaries.

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:

```
conda install SomePackage
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:

```
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:

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

Code A.5: Check Version of Packages

Appendix B

A Note on Distributions

B.1 Normal Distrubution

B.2 Bernoulli's Distribution

Definition B.2.1 (**Bernoulli distribution**)
The **Bernoulli distribution**, named after Swiss mathematician Jacob Bernoulli, *is the discrete probability distribution of a random variable which takes the value 1 with probability p and the value* 0 *with probability* $q = 1 - p$.

The Bernoulli distribution is a special case of the binomial distribution where a single trial is conducted (so n would be 1 for such a binomial distribution). It is *also a special case of the two-point distribution, for which the possible outcomes need not be 0 and 1.*

If X is a random variable with a Bernoulli distribution, then:

$$P(X = 1) = p$$
$$P(X = 0) = q = 1 - p$$

The Bernoulli distrubution is simply $B(1, p)$, also written as Bernoulli(p).