Chapter 1

Training Simple Machine Learning Algorithms for Classification

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

1.1 Introduction

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

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

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

The goals of this chapter are the following:

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

1.2 Artificial Neurons: Early Stages of Machine Learning

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

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

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

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

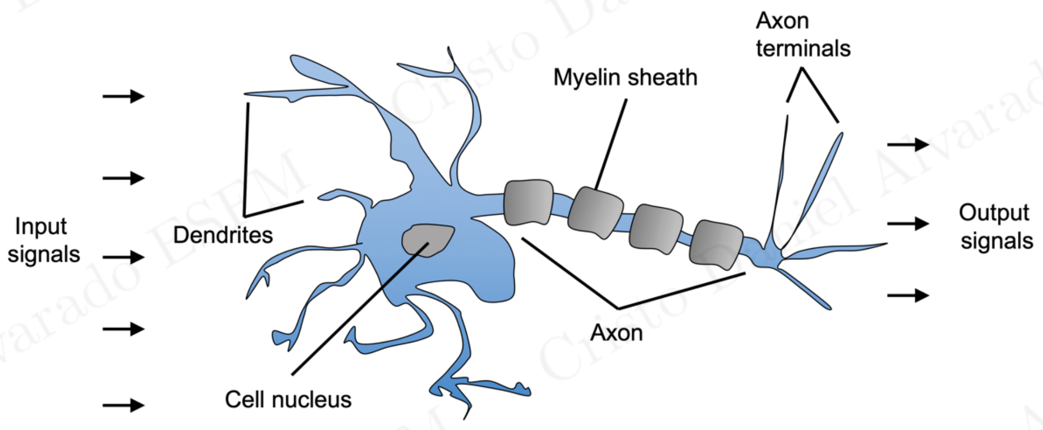


Figure 1.1: Biological Neuron.

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

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

1.3 Formal Definition of an Artificial Neuron

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

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

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

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

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

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

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

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

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

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

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

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

where:

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

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

1.4 Perceptron Learning Rule

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

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

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

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

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

and,

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

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

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

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

and,

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

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

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

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

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

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

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

2. **When prediction is correct:**

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

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

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

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

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

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

5. The weight update Δw_j is proportional to the value of $x_j^{(i)}$. For instance, if we have another example $x_j^{(i)} = 2$ that is incorrectly classified as class 0, we will push the decision boundary by an even larger extent to classify this example correctly the next time:
   $$\Delta w_j = (1-0) \times 2 = 2$$
   $$\Delta b = (1-0) = 1$$

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

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

1.5 Implementing a Perceptron Learning Algorithm in Python

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

1.6 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 1.6.1
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

        """
```

```python
        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 1.1: 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 1.6.1 (**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 1.6.2 (**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 1.6.2 (**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 1.6.3 (**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 1.6.4 (**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 1.6.1
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 1.6.5 (**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 1.6.1
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.*

1.7 Training of the Perceptron Algotithm on the Iris Dataset

Observation 1.7.1 (**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 1.7.1 (**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.*

1.7.1 OvA method for multi-class classification

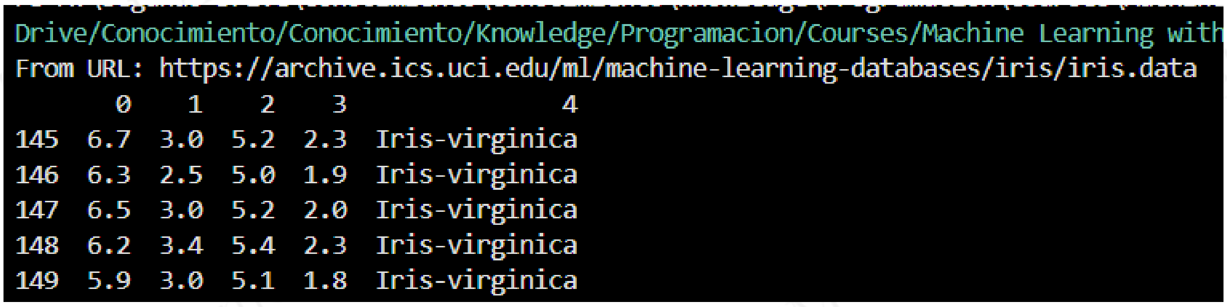
We can extend any binary classifier to a multi-class problems using the method called **OvA** or **one-versus-rest** (**OvR**).

Definition 1.7.1 (**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*:

```python
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 1.2: 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 1.2: 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*:

```python
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 1.3: Extraction and Visualization of Iris Dataset.

Observation 1.7.2 (Use of the Code)
Add this code to the remaining code.

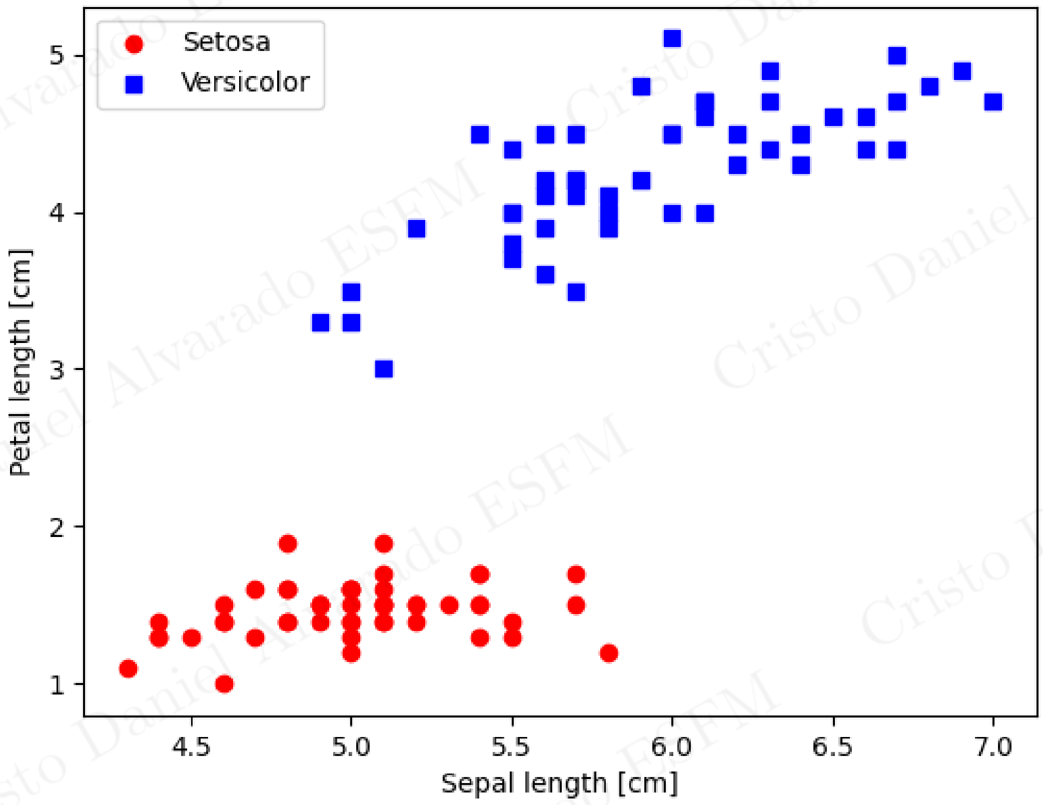


Figure 1.3: Plot of the Seritosa and Versicolor on the Iris Dataset.

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

Observation 1.7.3
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.

1.7.2 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')
```

```python
plt.show()
```
Code 1.4: Training the Perceptron Algorithm with the Iris Dataset

Observation 1.7.4
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.

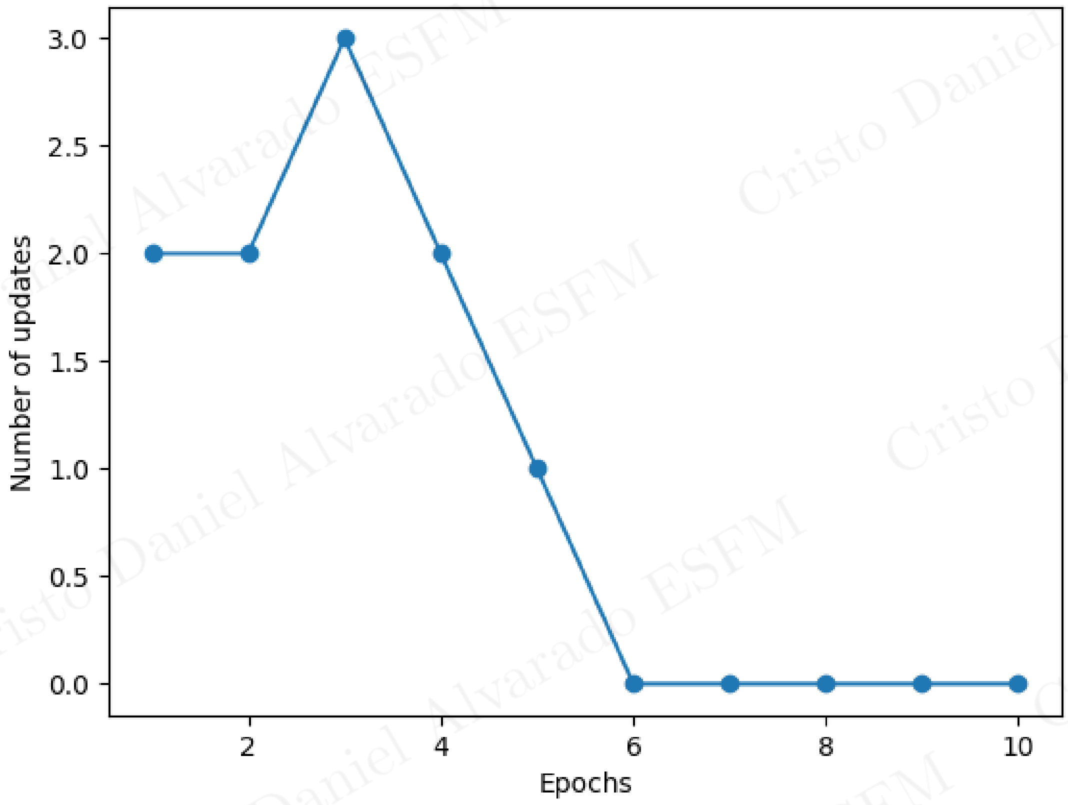


Figure 1.4: Epoch Plot and Number of Updates on the Iris Dataset.

Observation 1.7.5 (**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*:

```python
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 1.5: 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 1.6: 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 1.7.6 (**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 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

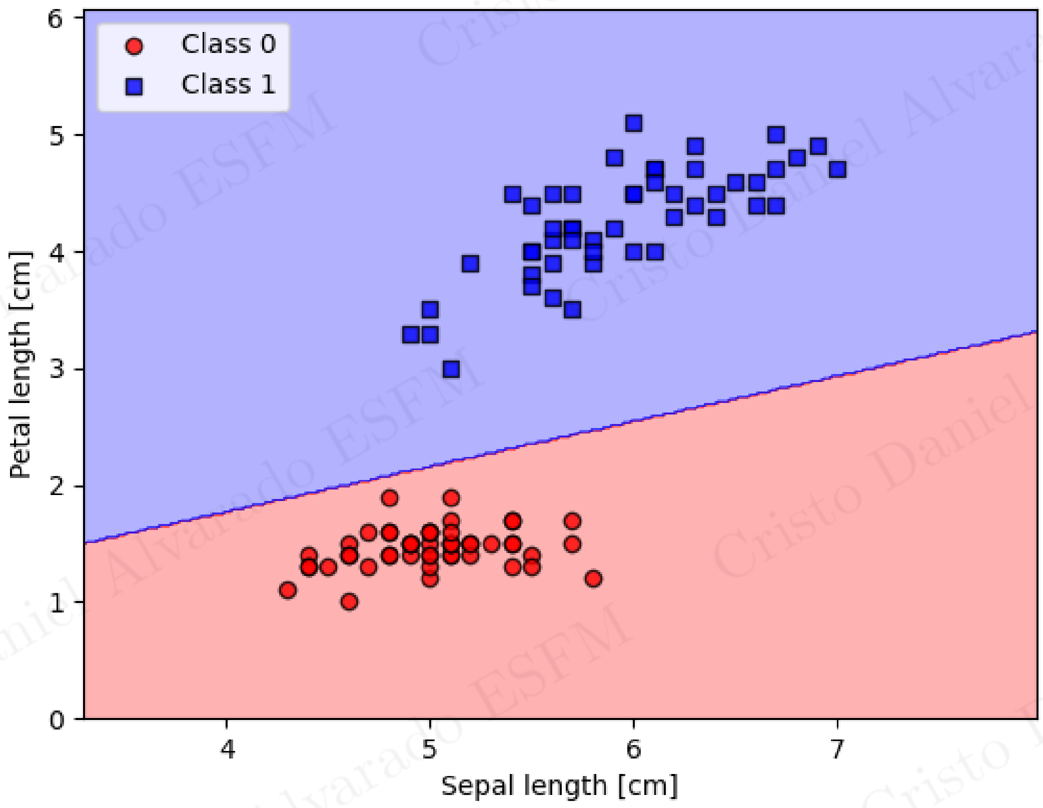


Figure 1.5: Perceptron Visual Classification on the Iris Dataset.

of the proof in the lecture notes at Perceptron Slides.

1.8 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 1.8.1 (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 1.8.1 (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 1.8.2
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.

1.8.1 Main Differenes Between Perceptron and Adaline

Observation 1.8.3 (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.

1.8.2 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 1.8.4 (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. Also, this function is **convex**.

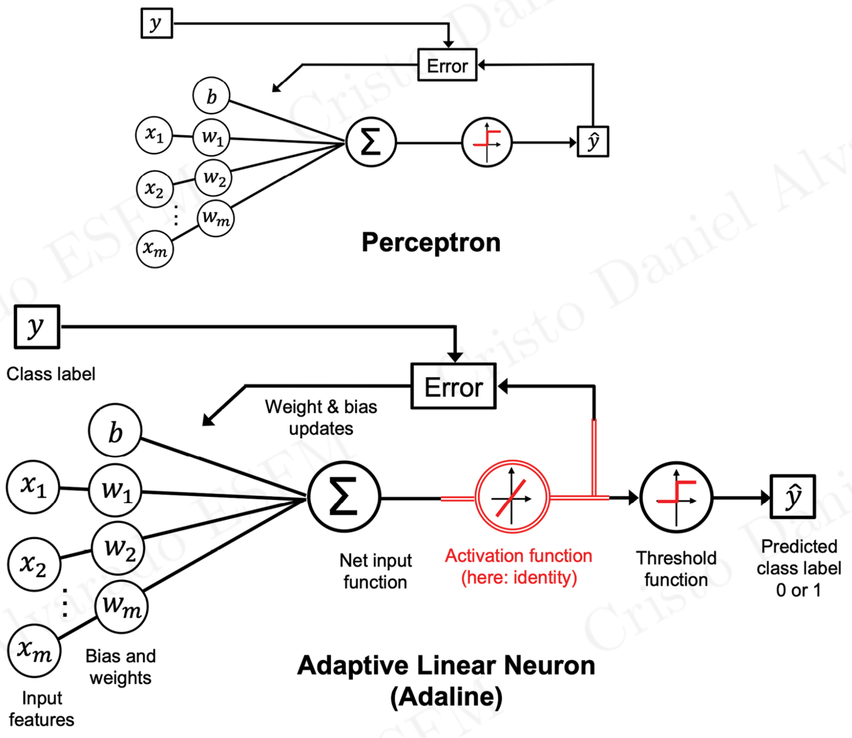


Figure 1.6: Perceptron vs Adeline Work Flux.

Definition 1.8.1 (**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:

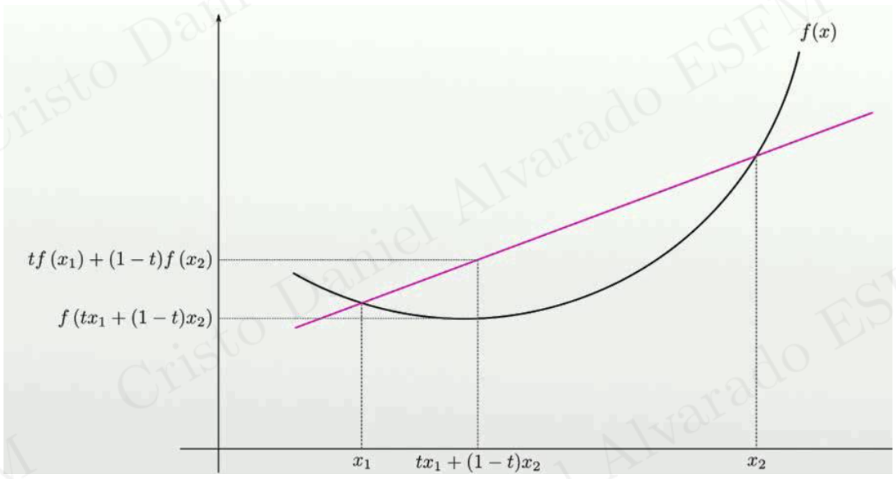


Figure 1.7: Convex Function.

Observation 1.8.5 (**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 1.8.2 (**Gradient Descent**)
The idea behind **gradient descent** is to *climb down a hill until a local or global loss minimum is reached.*

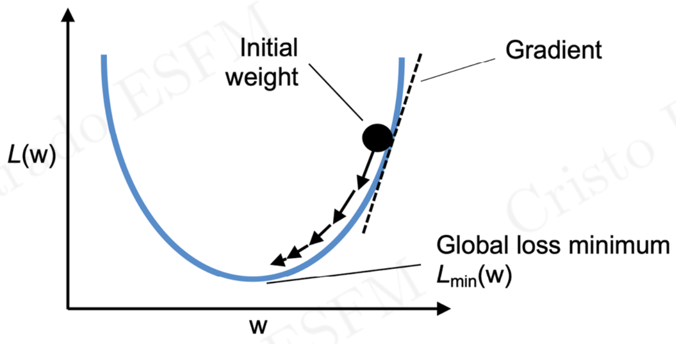


Figure 1.8: Gradient Descent.

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 1.8.6 (**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 1.8.7
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 1.8.8 (**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**.

1.9 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*:

```python
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
```

```python

    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 1.7: Adaline Implementation in Python.

Observation 1.9.1 (**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:

```python
for w_j in range(self.w_.shape[0]):
    self.w_[w_j] += self.eta * \
        (2.0 * (X[:, w_j]*errors)).mean()
```
Code 1.8: 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:

```python
self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
```
Code 1.9: Weight Update Optimization.

Observation 1.9.2 (**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 1.9.3 (**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 1.9.4 (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.

1.9.1 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 1.10: 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.*

1.10 Improving Gradient Descent Through Feature Scaling

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

Definition 1.10.1 (**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 1.10.1 (**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 1.10.1 (**Feature Scaling**)
Gradiant descent is one of the many algorithms that benefit from **feature scaling**.

Definition 1.10.2 (**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 1.10.2 (**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 1.10.2 (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**).

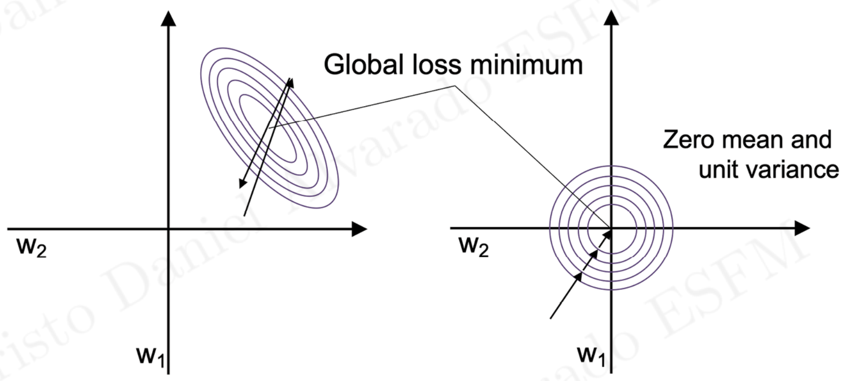


Figure 1.9: Global Loss Minimum with and without Zero Variance.

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 1.11: 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 1.12: Adaline With Standarization Procedure

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

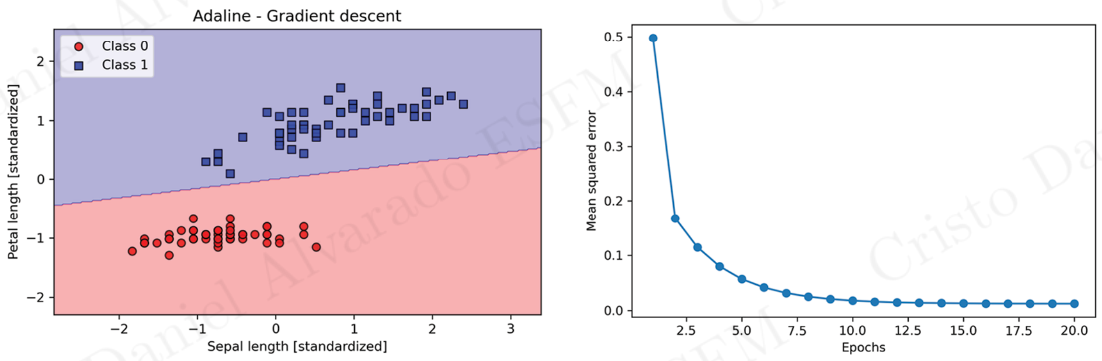


Figure 1.10: Adeline after Standarization Procedure

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

1.10.1 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 1.10.3 (**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 1.10.4
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 1.10.5 (**Note**)
Use an adaptive learning rate for better training results

Idea 1.10.1 (**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 1.10.6 (**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
```

```python
    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])
    self.losses_ = []
    for i in range(self.n_iter):
        if self.shuffle:
            X, y = self._shuffle(X, y)
        losses = []
        for xi, target in zip(X, y):
            losses.append(self._update_weights(xi, target))
        avg_loss = np.mean(losses)
        self.losses_.append(avg_loss)
    return self

def partial_fit(self, X, y):
    """Fit training data without reinitializing the weights"""
    if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Shuffle training data"""
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights to small random numbers"""
        self.rgen = np.random.RandomState(self.random_state)
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01,
                                    size=m)
        self.b_ = np.float_(0.)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Apply Adaline learning rule to update the weights"""
        output = self.activation(self.net_input(xi))
        error = (target - output)
        self.w_ += self.eta * 2.0 * xi * (error)
        self.b_ += self.eta * 2.0 * error
        loss = error**2
        return loss

    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 1.13: 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 1.10.7
Display the plot with labeled axes using `plt.show()`.

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

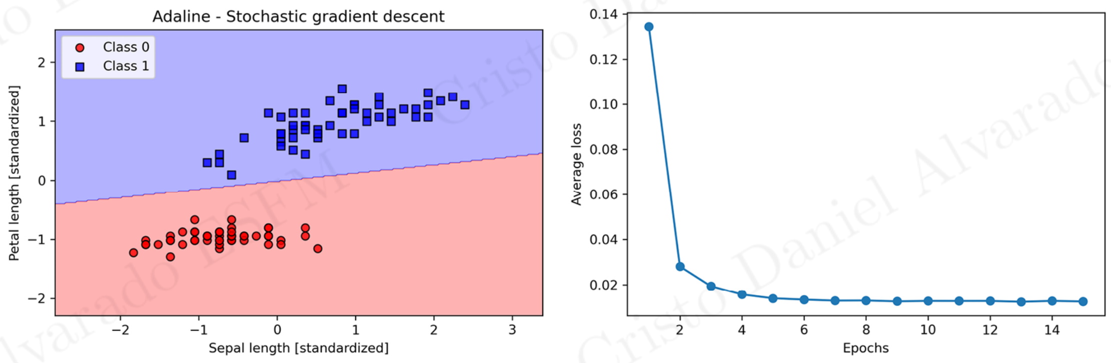


Figure 1.11: Adaline Implementation using Stochastic Gradient DEscent

Idea 1.10.2 (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.