

Lecture 3-1

Machine Learning -Introduction

Three different types of machine learning

Supervised Learning

- Labeled data
- Direct feedback
- Predict outcome/future

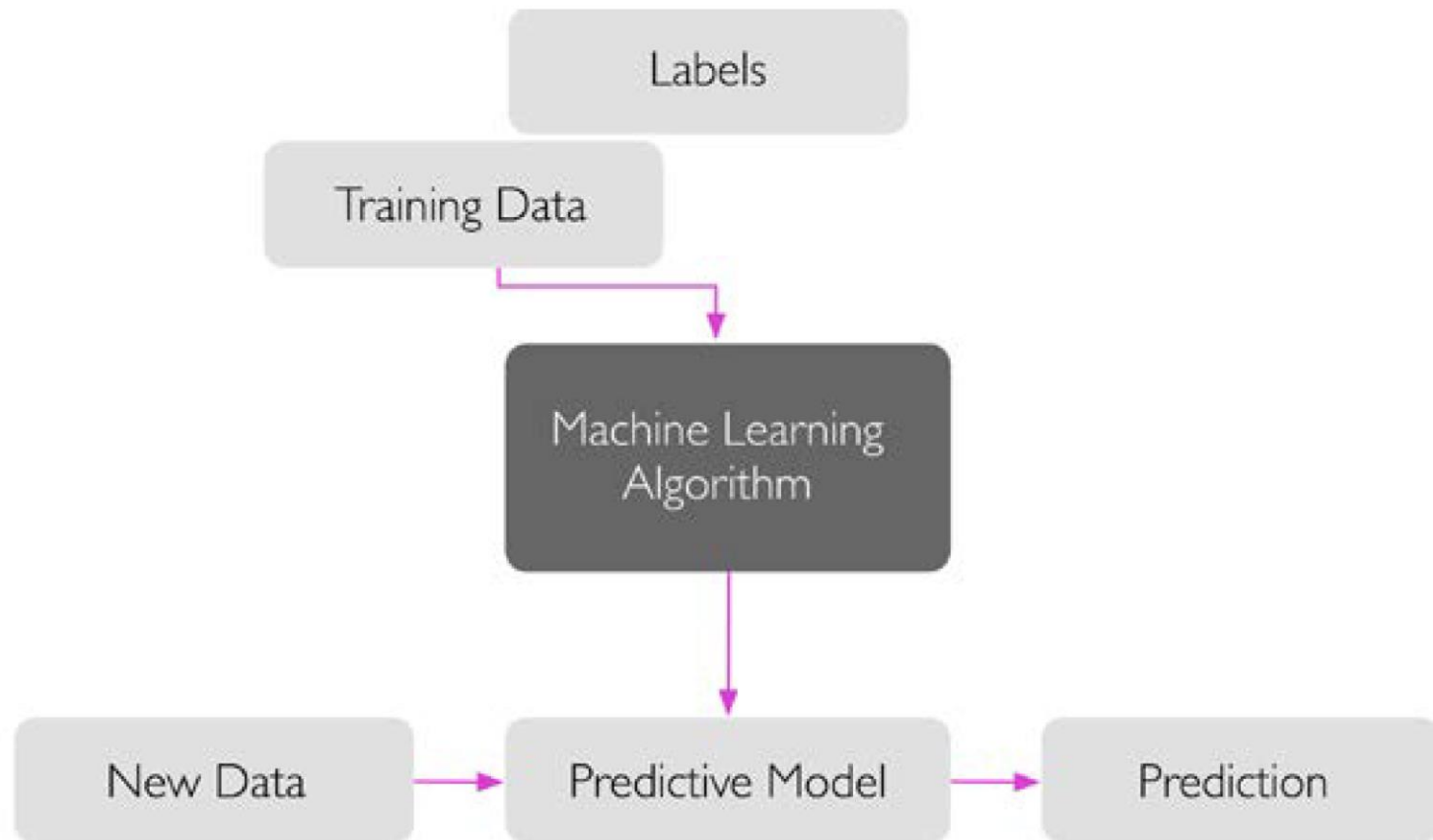
Unsupervised Learning

- No labels
- No feedback
- Find hidden structure in data

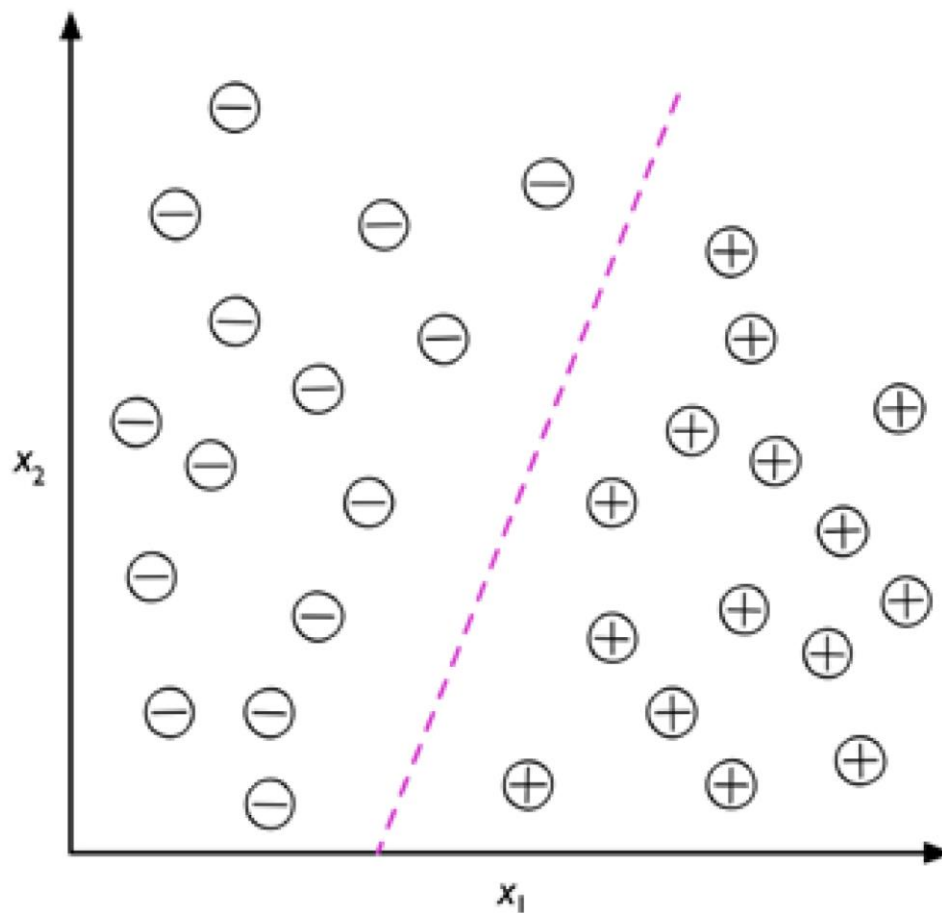
Reinforcement Learning

- Decision process
- Reward system
- Learn series of actions

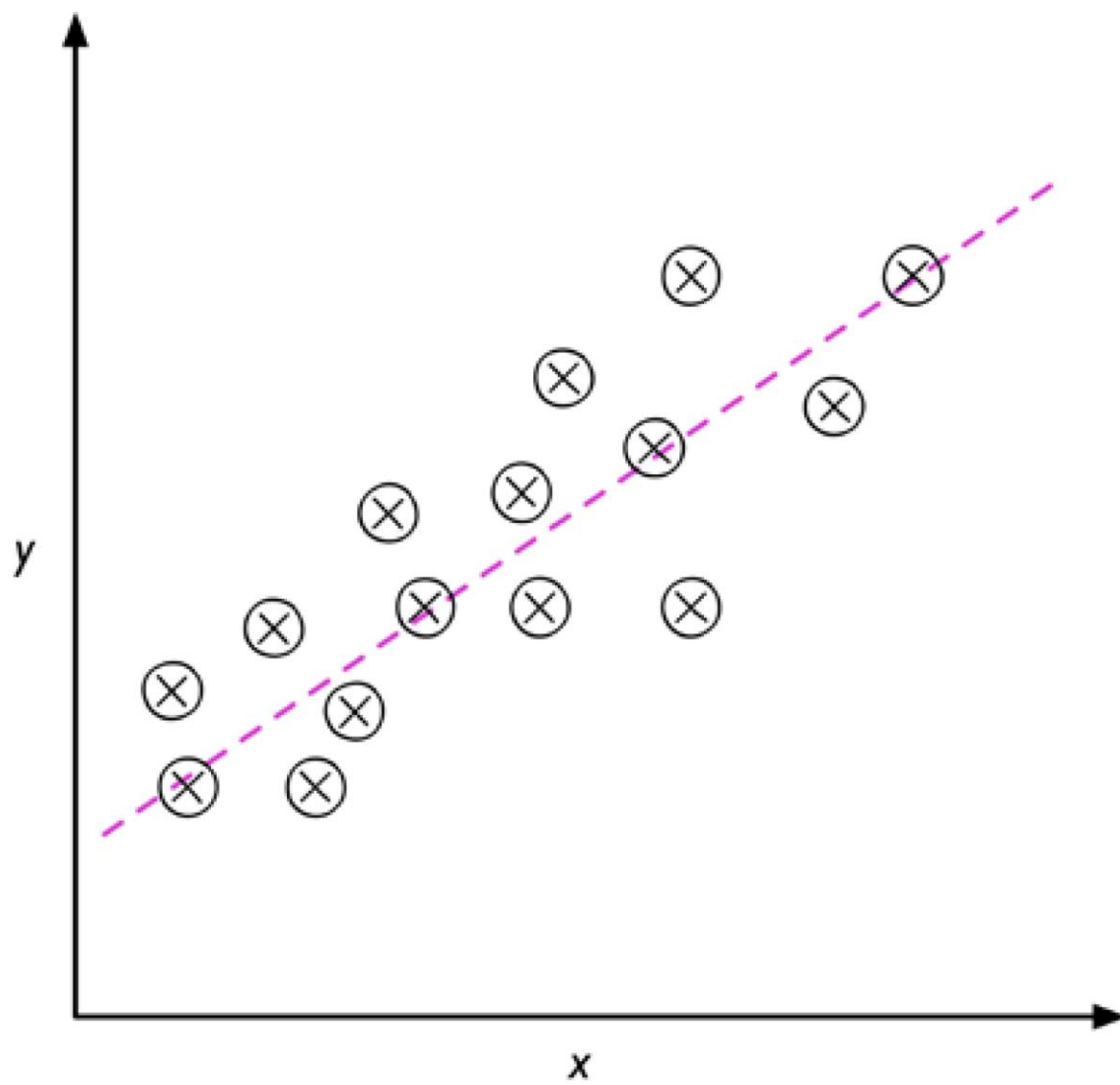
Making predictions about the future with supervised learning



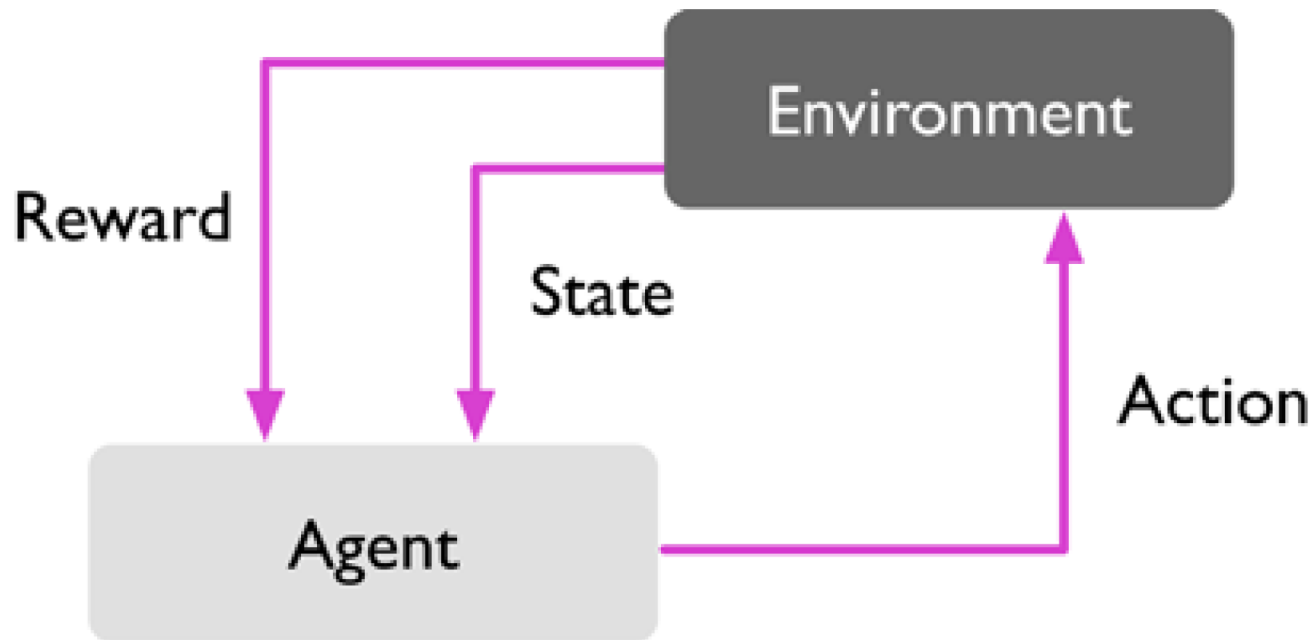
Classification for predicting class labels



Regression for predicting continuous outcomes



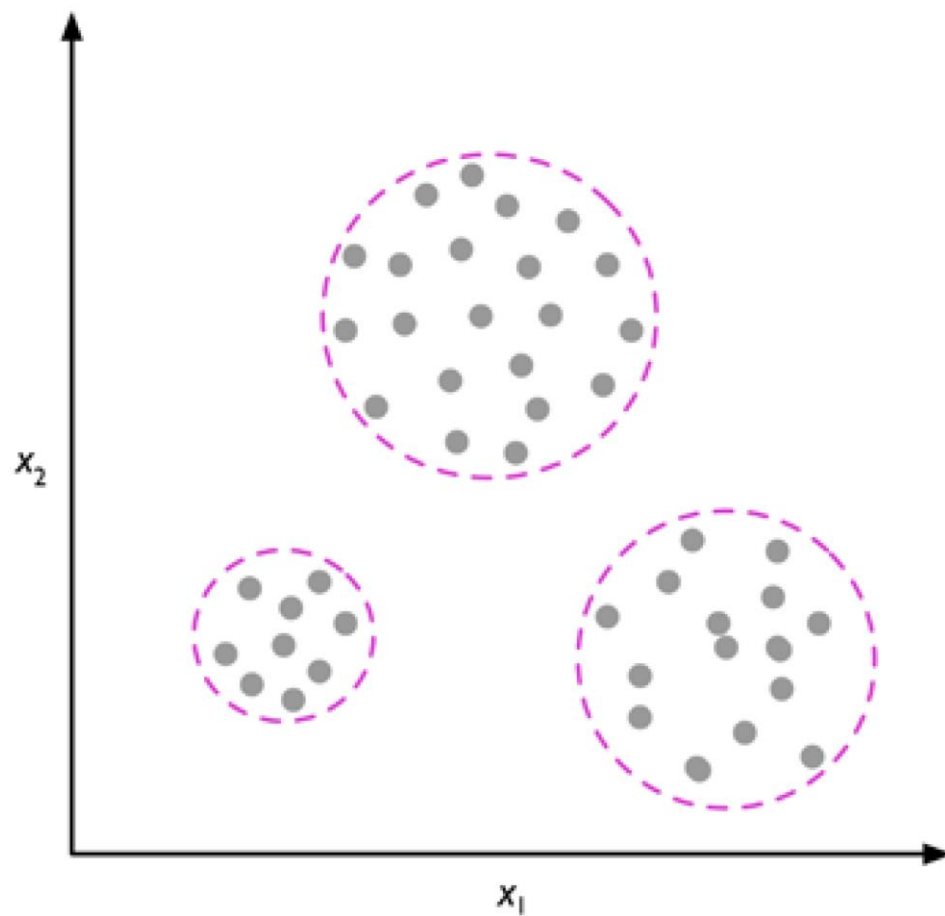
Solving interactive problems with reinforcement learning



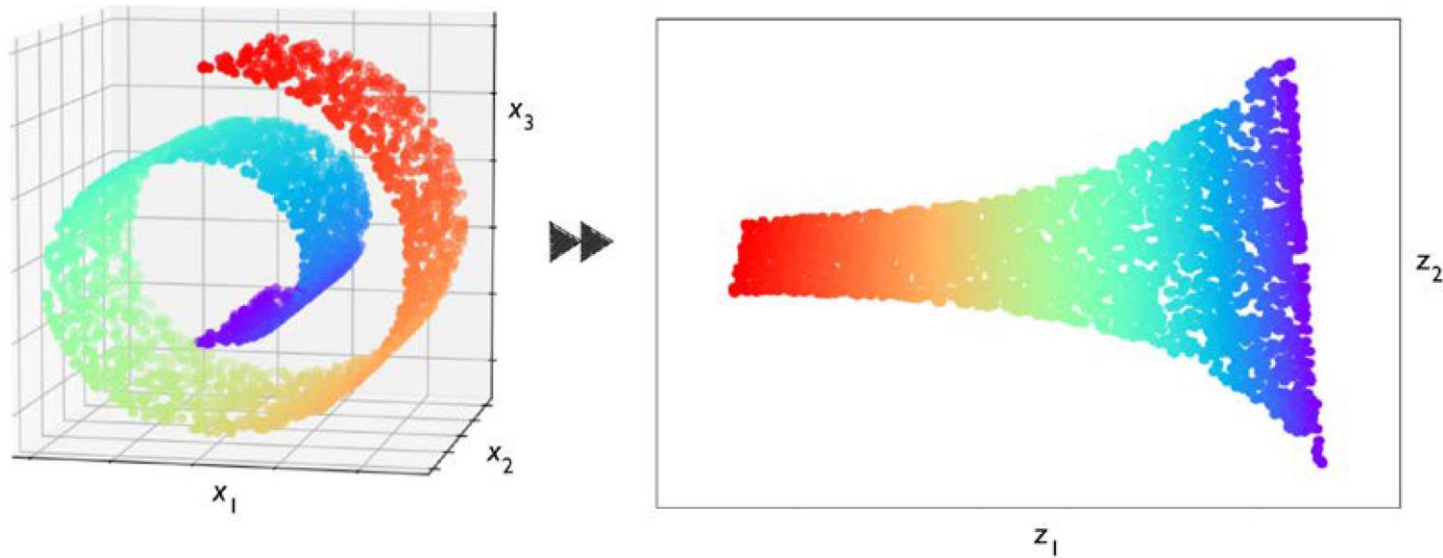
Discovering hidden structures with unsupervised learning

- In supervised learning, we know the right answer beforehand when we train a model, and in reinforcement learning, we define a measure of reward for particular actions carried out by the agent.
- In unsupervised learning, however, we are dealing with unlabeled data or data of unknown structure.
- Using unsupervised learning techniques, we are able to explore the structure of our data to extract meaningful information without the guidance of a known outcome variable or reward function.

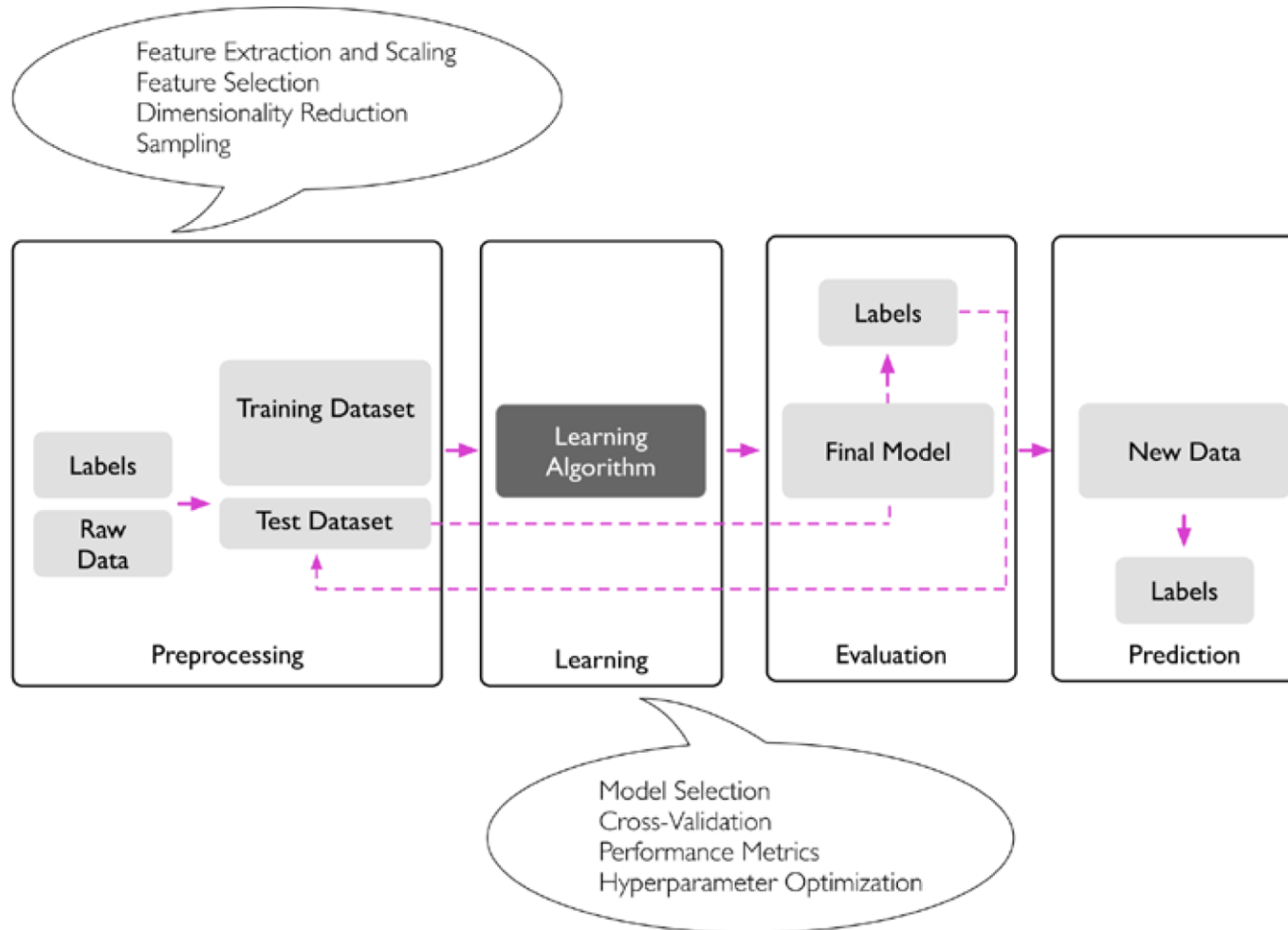
Finding subgroups with clustering



Dimensionality reduction for data compression



A roadmap for building machine learning systems



Summary

- In this chapter, we explored machine learning at a very high level and familiarized ourselves with the big picture and major concepts.
- We learned that supervised learning is composed of two important subfields: classification and regression.
- While classification models allow us to categorize objects into known classes, we can use regression analysis to predict the continuous outcomes of target variables.
- Unsupervised learning not only offers useful techniques for discovering structures in unlabeled data, but it can also be useful for data compression in feature preprocessing steps.
- We briefly went over the typical roadmap for applying machine learning to problem tasks

References

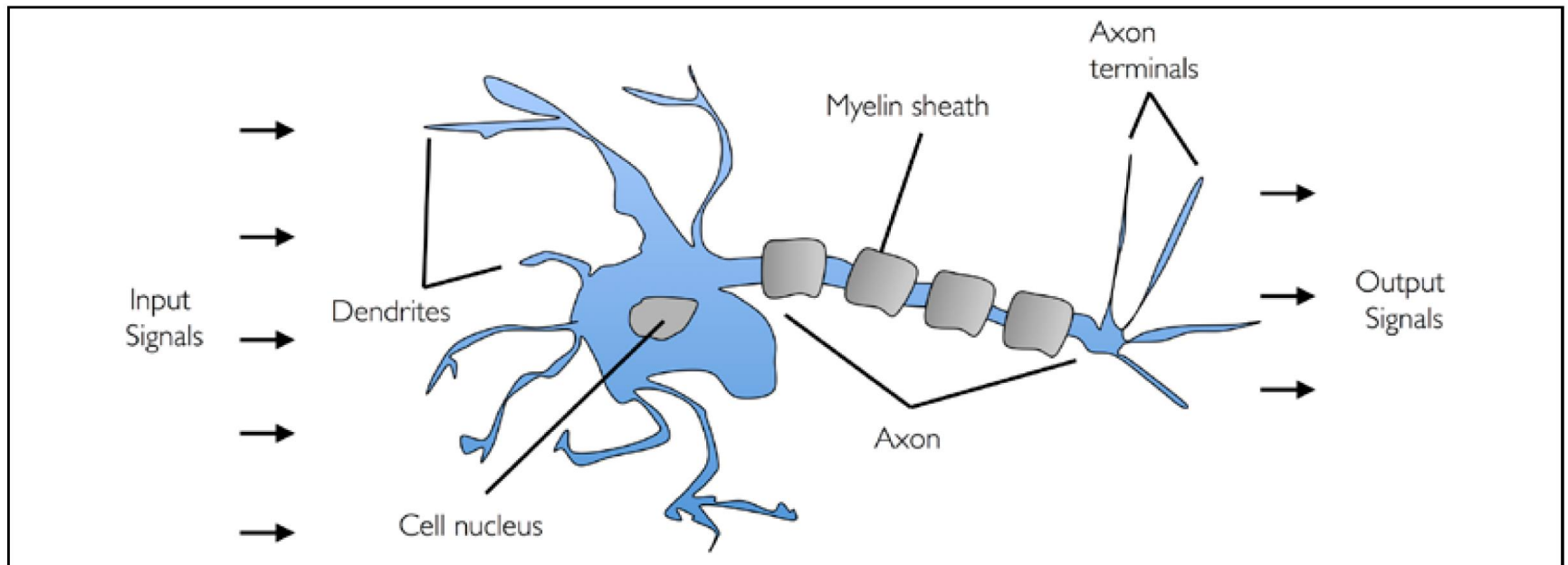
Python Machine Learning – Machine Learning and Deep Learning with Python, Scikit-learn, and TensorFlow 2, Third Edition, Packt Publishing

Lecture 3-2

Machine Learning

-Training Simple Machine Learning Algorithms for Classification

Artificial neurons – a brief glimpse into the early history of machine learning



McCulloch-Pitts (MCP) neuron, in 1943 (A Logical Calculus of the Ideas Immanent in Nervous Activity, W. S. McCulloch and W. Pitts, Bulletin of Mathematical Biophysics, 5(4): 115-133, 1943).

Biological neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals

Artificial neurons – a brief glimpse into the early history of machine learning

McCulloch and Pitts described such 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 signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

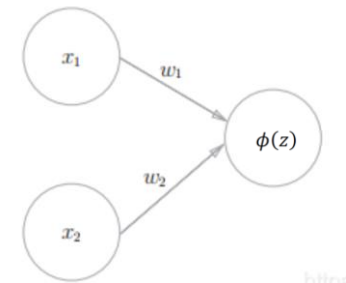
Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (*The Perceptron: A Perceiving and Recognizing Automaton*, F. Rosenblatt, Cornell Aeronautical Laboratory, 1957).

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.

The formal definition of an artificial neuron

More formally, we can put the idea behind **artificial neurons** into the context of a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. We can then define a decision function ($\phi(z)$) that takes a linear combination of certain input values, \mathbf{x} , and a corresponding weight vector, \mathbf{w} , where z is the so-called net input $z = w_1x_1 + w_2x_2 + \dots + w_mx_m$:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$



Now, if the net input of a particular example, $\mathbf{x}^{(i)}$, is greater than a defined threshold, θ , we predict class 1 , and class -1 otherwise. In the perceptron algorithm, the decision function, $\phi(\cdot)$, is a variant of a **unit step function**:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise.} \end{cases}$$

The formal definition of an artificial neuron

For simplicity, we can bring the threshold, θ , to the left side of the equation and define a weight-zero as $w_0 = -\theta$ and $x_0 = 1$ so that we write z in a more compact form:

$$z = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \mathbf{w}^T \mathbf{x}$$

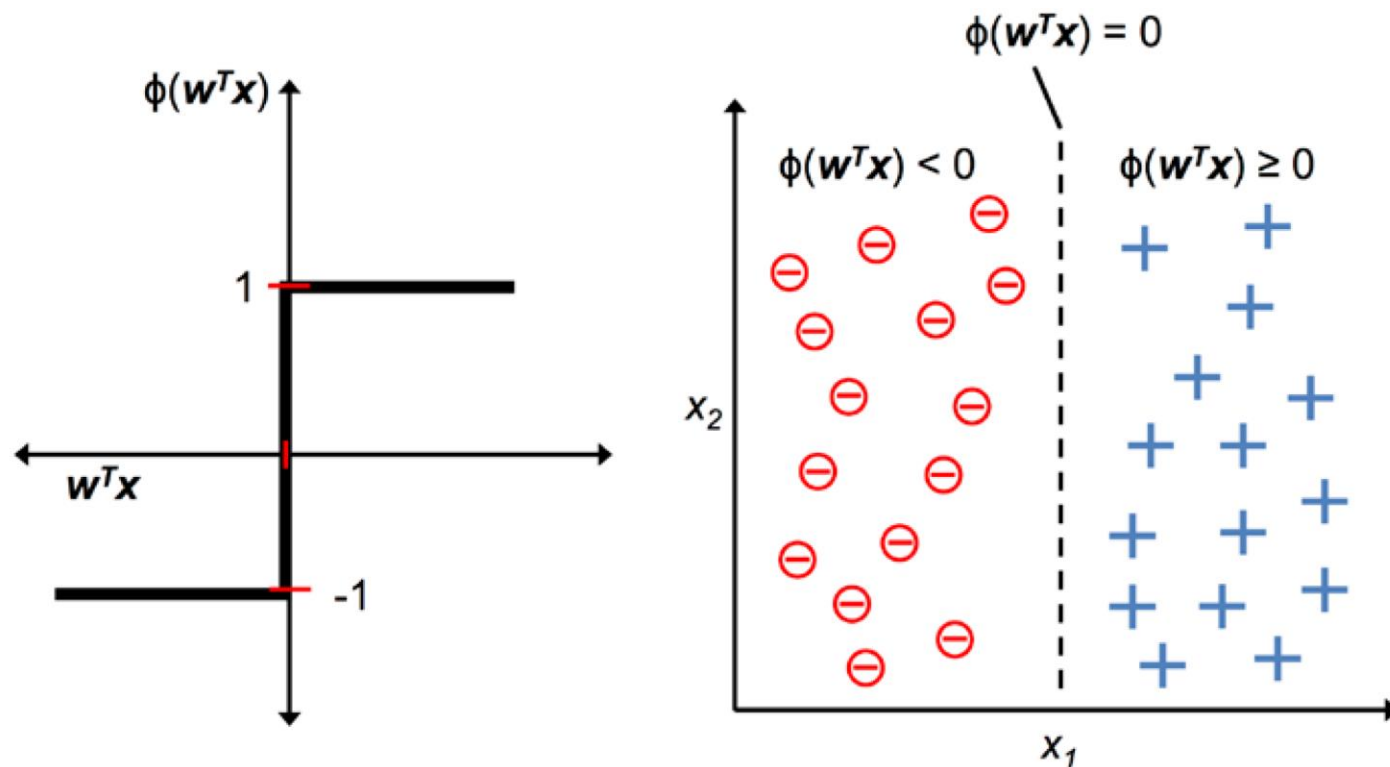
And:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

In machine learning literature, the negative threshold, or weight, $w_0 = -\theta$, is usually called the **bias unit**.

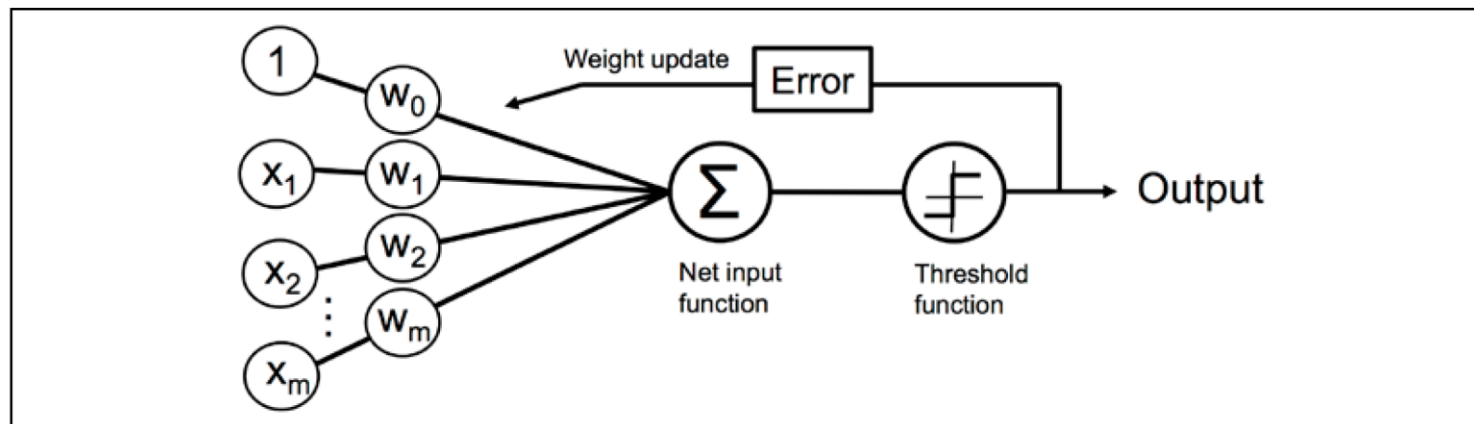
The formal definition of an artificial neuron

The following figure illustrates how the net input $z = \mathbf{w}^T \mathbf{x}$ is squashed into a binary output (-1 or 1) by the decision function of the perceptron (left subfigure) and how it can be used to discriminate between two linearly separable classes (right subfigure):



The perceptron learning rule

Now, before we jump into the implementation in the next section, what you just learned can be summarized in a simple diagram that illustrates the general concept of the perceptron:



The preceding diagram illustrates how the perceptron receives the inputs of an example, \mathbf{x} , and combines them with the weights, \mathbf{w} , to compute the net input. The net input is then passed on to the threshold function, which generates a binary output of -1 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.

Implementing a perceptron learning algorithm in Python

-An object oriented perceptron API

We will take an object-oriented approach to defining the perceptron interface as a Python class, which 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. As a convention, we append an underscore (`_`) to attributes that are not created upon the initialization of the object, but we do this by calling the object's other methods, for example, `self.w_`.

Implementing a perceptron learning algorithm in Python

The following is the implementation of a perceptron in Python:

```
import numpy as np

class Perceptron(object):
    """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.
    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
```

Implementing a perceptron learning algorithm in Python

```
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=1 + X.shape[1])
    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_[1:] += update * xi
            self.w_[0] += 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_[1:]) + self.w_[0]

def predict(self, X):
    """Return class label after unit step"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Implementing a perceptron learning algorithm in Python

Using this perceptron implementation, we can now initialize new Perceptron objects with a given learning rate, η , and the number of epochs, `n_iter` (passes over the training dataset).

Via the `fit` method, we initialize the weights in `self.w_` to a vector, \mathbb{R}^{m+1} , where m stands for the number of dimensions (features) in the dataset, and we add 1 for the first element in this vector that represents the bias unit. Remember that the first element in this vector, `self.w_[0]`, represents the so-called bias unit that we discussed earlier.

Also notice that this vector contains small random numbers drawn from a normal distribution with standard deviation 0.01 via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, where `rgen` is a NumPy random number generator that we seeded with a user-specified random seed so that we can reproduce previous results if desired.

It is important to keep in mind that we don't initialize the weights to zero because the learning rate, η (η), only has an effect on the classification outcome if the weights are initialized to non-zero values. If all the weights are initialized to zero, the learning rate parameter, η , affects only the scale of the weight vector, not the direction. If you are familiar with trigonometry, consider a vector, $v_1 = [1 \ 2 \ 3]$, where the angle between v_1 and a vector, $v_2 = 0.5 \times v_1$, would be exactly zero, as demonstrated by the following code snippet:

Implementing a perceptron learning algorithm in Python

```
>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...                      np.linalg.norm(v2)))
0.0
```

Here, `np.arccos` is the trigonometric inverse cosine, and `np.linalg.norm` is a function that computes the length of a vector (our decision to draw the random numbers from a random 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).

Implementing a perceptron learning algorithm in Python

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 that we discussed in the previous section.

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 we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the `self.errors_` list so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product, $\mathbf{w}^T \mathbf{x}$.

Training a perceptron model on the Iris dataset

To test our perceptron implementation, we will restrict the following analyses and examples in the remainder of this chapter to two feature variables (dimensions). Although the perceptron rule is not restricted to two dimensions, considering only two features, sepal length and petal length, will allow us to visualize the decision regions of the trained model in a scatter plot for learning purposes.

Note that 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— for example, the one-vs.-all (OvA) technique.

Training a perceptron model on the Iris dataset

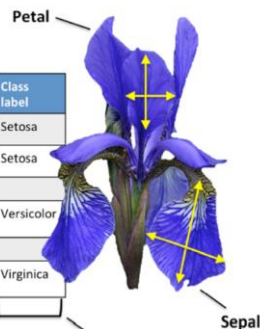
First, we will use the pandas library to load the Iris dataset directly from the UCI Machine Learning Repository into a DataFrame object and print the last five lines via the tail method to check that the data was loaded correctly:

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

Samples
(instances, observations)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...
50	6.4	3.5	4.5	1.2	Versicolor
...
150	5.9	3.0	5.0	1.8	Virginica

Features
(attributes, measurements, dimensions)



Petal
Sepal
Class labels
(targets)

	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

Training a perceptron model on 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 -1 (setosa), that we assign to a vector, y , where the values method of a pandas DataFrame yields the corresponding NumPy representation.

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 can 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', -1, 1)

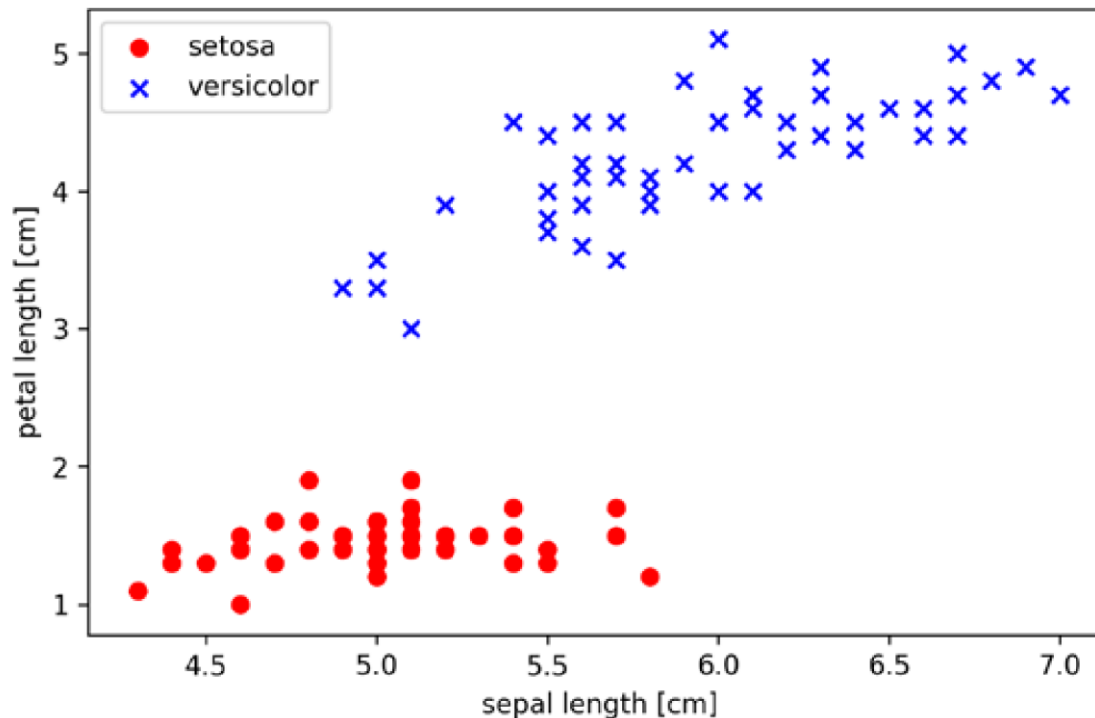
>>> # extract sepal length and petal length
>>> X = df.iloc[0:100, [0, 2]].values

>>> # 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='x', label='versicolor')
```

Training a perceptron model on the Iris dataset

```
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should now see the following scatterplot:



Training a perceptron model on the Iris dataset

The preceding scatterplot shows the distribution of flower examples in the Iris dataset along the two feature axes: petal length and sepal length (measured in centimeters). In this two-dimensional feature subspace, we can see that a linear decision boundary should be sufficient to separate Setosa from Versicolor flowers.

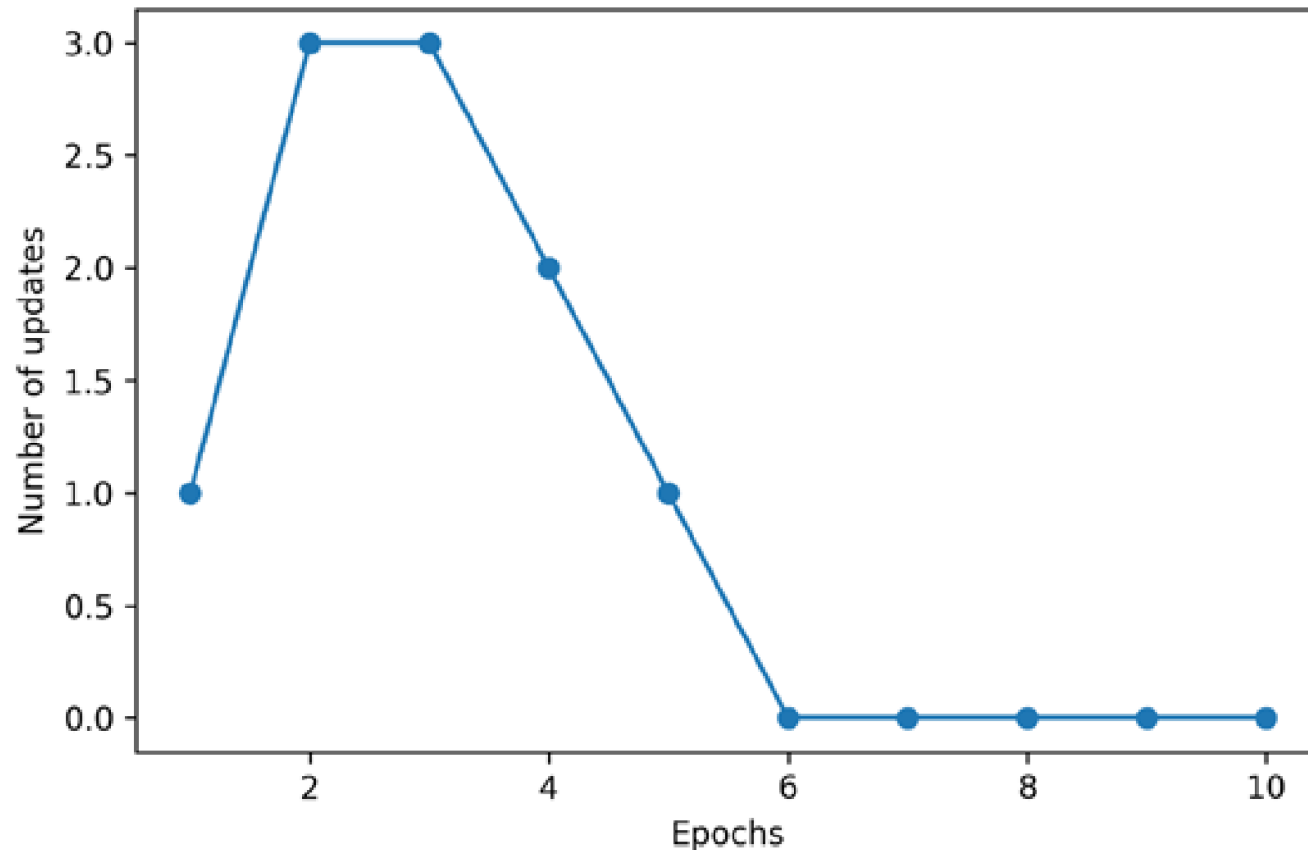
Thus, a linear classifier such as the perceptron should be able to classify the flowers in this dataset perfectly.

Now, it's time to train our perceptron algorithm on the Iris data subset that we just extracted. Also, we will 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:

```
>>> 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()
```

Training a perceptron model on the Iris dataset

After executing the preceding code, we should see the plot of the misclassification errors versus the number of epochs, as shown in the following graph:



As we can see in the preceding plot, our perceptron converged after the sixth epoch and should now be able to classify the training examples perfectly.

Training a perceptron model on the Iris dataset

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 = ('s', 'x', 'o', '^', '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))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, 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],
```


Training a perceptron model on the Iris dataset

```
y=X[y == c1, 1],  
alpha=0.8,  
c=colors[idx],  
marker=markers[idx],  
label=c1,  
edgecolor='black')
```

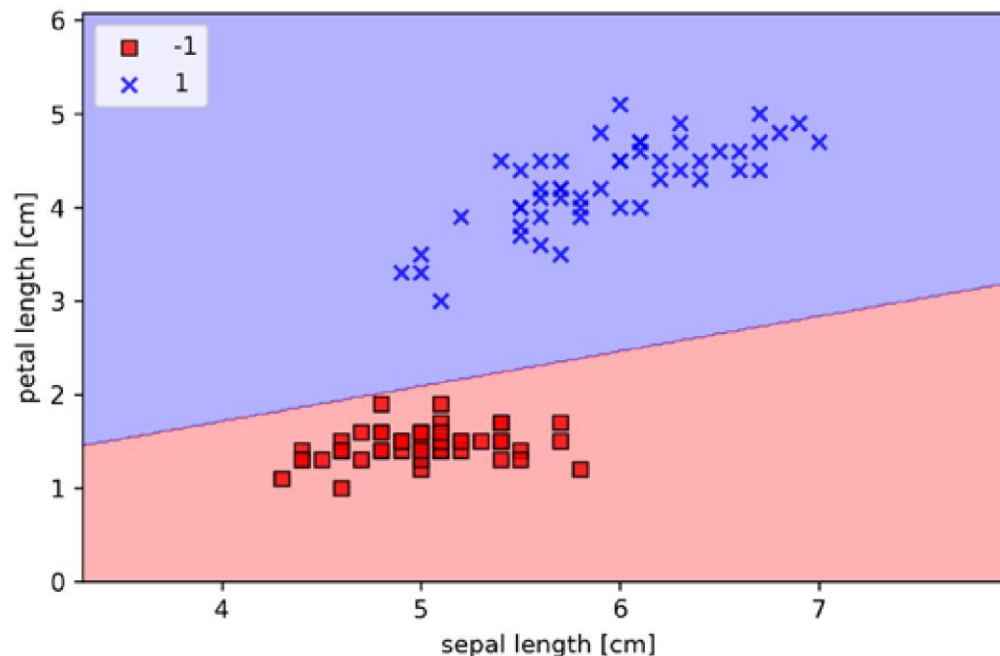
First, we define a number of colors and markers and create a colormap from the list of colors via ListedColormap. Then, we determine the minimum and maximum values for the two features and use those feature vectors to create a pair of grid arrays, `xx1` and `xx2`, via the NumPy `meshgrid` function. Since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

After reshaping the predicted class labels, `Z`, into a grid with the same dimensions as `xx1` and `xx2`, we can now 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:

Training a perceptron model on the Iris dataset

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should now see a plot of the decision regions, as shown in the following figure:



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

References

Python Machine Learning – Machine Learning and Deep Learning with Python, Scikit-learn, and TensorFlow 2, Third Edition, Packt Publishing